Issue 1
October 1984

AT&T 3B2 Computer Utilities

Select Code 305-408

Comcode 403778327

NOTE

Two UTILITIES binders are provided with each 3B2 Computer. One is intended for filing the *System Administration Utilities Guide.* The other is intended for filing all provided and optional Utilities Guides.

All the Utilities Guides come with tab separators and the pages are prepunched for easy filing in the binders.

The comment forms in this binder may be used for any comments concerning the 3B2 Computer Utilities Guides, Software Information Bulletins (SIB), or Option Manuals. When commenting on a document, please enter the document title and the six-digit select code on the comment form.

Additional copies of this binder may be ordered. Follow the ordering procedures specified in the *Documentation Catalog* provided with your 3B2 Computer.

Issue 1
October 1984

AT&T 3B2 Computer UNIX™ System V Release 2.0 Essential Utilities Software Information Bulletin

Select Code
305-349

Comcode
403777907

NOTE

This Software Information Bulletin (SIB) should be filed in the *3B2 Computer Owner/Operator Manual.* A tab separator, labeled " SOFTWARE INFORMATION BULLETINS," has been placed at the back of the *Owner/Operator Manual* in order to provide a convenient place for filing SIB's. Place the tab separator provided with this SIB in front of the title page and file this material behind the SOFTWARE INFORMATION BULLETINS tab separator in the *Owner/Operator Manual.*

ESSENTIAL SOFTWARE INFORMATION BULLETIN

INTRODUCTION

This Software Information Bulletin provides important information concerning the Essential Utilities. Please read this bulletin carefully before attempting to use these utilities.

The AT&T 3B2 Computer Essential Utilities support all other 3B2 Computer UNIX* System Utilities. The Essential Utilities are stored on hard disk along with the UNIX System. The Essential Utilities consist of three general types of commands:

1. General Commands - These commands support all basic UNIX System operations.

2. Simple Administration Command - This command and associated subcommands enable a user to easily perform the initial setup of the 3B2 Computer. Simple Administration also provides the necessary tasks for an unsophisticated UNIX System user to administer the file structure on the 3B2 Computer hard disk.

3. System Administration Commands - These commands are provided to support Simple Administration software. They can be accessed by users familiar with "raw UNIX System" administration and are typically used with commands provided as part of System Administration Utilities.

* Trademark of AT&T Bell Laboratories

The commands that comprise Essential Utilities are listed below. Those commands that are considered system administration are flagged with a circumflex (ˆ).

bcheckrc ˆ	fsck ˆ	mount ˆ	setup
brc ˆ	fsstat ˆ	mountall ˆ	sh
cat	getmajor ˆ	mountfsys ˆ	shutdown ˆ
cd	getopt	mv	sleep
checkfsys ˆ	getty ˆ	newboot ˆ	sort
chgrp	grep	newgrp ˆ	stty
chmod	hdeadd ˆ	news	su ˆ
chown	hdefix ˆ	nkill	sync ˆ
clri ˆ	hdelogger ˆ	passwd	sysadm
cmp	id ˆ	powerdown ˆ	tee
cp	init ˆ	pr	telinit ˆ
cpio	kill	prtvtoc ˆ	test
cron ˆ	killall ˆ	ps	touch
date	labelit ˆ	pump ˆ	true
dd ˆ	ln	pwd	u3b2 (machid)
devnm ˆ	logdir	rc ˆ	uadmin ˆ
df ˆ	login	rc0 ˆ	umask
diff	ls	rc2 ˆ	umount ˆ
drvinstall ˆ	mail	red	umountall ˆ
du ˆ	mailx	rm	uname
echo	makefsys ˆ	rmail	umountfsys ˆ
ed	mesg	rmdir	wait
errdump ˆ	mkboot ˆ	rsh	wall
expr	mkdir	sanityck ˆ	wc
false	mkfs ˆ	sed	who
fmtflop ˆ	mknod ˆ	setclk ˆ	write
fmthard ˆ	mkunix ˆ	setmnt ˆ	

SOFTWARE DEPENDENCIES

The Essential Utilities are independent of all optional utilities. No other software need be installed to use the Essential Utilities.

NOTES ON USING UTILITIES

Any user can access the Essential Utilities commands. You do not have to be logged in as root.

DOCUMENTATION

This Software Information Bulletin should be inserted in the *3B2 Computer Owner/Operator Manual*.

The most commonly used Essential Utilities general commands are described in the *UNIX System V User Guide*. Descriptions of the Simple Administration command and associated subcommands are located in the *3B2 Computer Owner/Operator Manual*. The system administration-type commands provided as part of the Essential Utilities are described in the *3B2 Computer System Administration Utilities Guide*.

RELEASE FORMAT

The Essential Utilities are provided on the hard disk of every 3B2 Computer. The commands of Essential Utilities are located in the /**bin** and the /**etc** directories.

A backup copy of the Essential Utilities exists on the five floppy disks labeled ''AT&T 3B2 Essential Utilities (Part _ of 5).'' These floppy disks are used to restore the hard disk with the UNIX Operating System and Essential Utilities if they ever become corrupted or are destroyed. Procedures to restore hard disk information from floppy disks are found in the *3B2 Computer System Administration Utilities Guide*.

This update inventory should be placed behind the *3B2 Computer Directory and File Management Utilities Guide* title page. This inventory identifies the pages that have been added, changed, or removed by the *3B2 Computer Directory and File Management Utilities Guide Update* (305-423), dated December 10, 1984.

Revised Pages	**Date**
Table of Contents, Chapter 2	12/10/84
2-2	12/10/84
2-3	12/10/84
2-9	12/10/84

Added Pages	**Date**
2-9a	12/10/84
2-9b	12/10/84
2-9c	12/10/84
2-9d	12/10/84
2-9e	12/10/84
2-9f	12/10/84
ar manual pages	11/84

Removed Pages	**Date**
2-69	12/10/84
2-70	12/10/84
2-71	12/10/84
2-72	12/10/84
2-73	12/10/84
find manual pages	10/84

AT&T

Issue 1
October 1984

AT&T 3B2 Computer UNIX™ System V Release 2.0 Directory and File Management Utilities Guide

Select Code
305-418

Comcode
403778491

TRADEMARKS

NOTICE

NOTE

This Utilities Guide contains descriptive information and UNIX* System manual pages for the commands included in one of the utilities provided with your 3B2 Computer. Since this utilities is provided with the 3B2 Computer, the manual pages have already been filed in the *3B2 Computer UNIX System V User Reference Manual.* If you do not need duplicate copies of these manual pages, they may be discarded.

A UTILITIES binder is provided with the 3B2 Computer for you to keep the descriptive information from all the Utilities Guides together. Remove the descriptive information from the soft cover, place the provided tab separator in front of the title page, and file this material in the UTILITIES binder. As previously mentioned, UNIX System manual pages may be destroyed.

If you ordered extra copies of this Utilities Guide, they should be left in the individual soft covers.

* Trademark of AT&T Bell Laboratories

Issue 2
December 1984

AT&T 3B2 Computer UNIX™ System V Release 2.0 Directory and File Management Utilities Software Information Bulletin

Select Code	Comcode
305-355	403778186

NOTE

This Software Information Bulletin (SIB) should be filed in the *3B2 Computer Owner/Operator Manual.* A tab separator, labeled " SOFTWARE INFORMATION BULLETINS," has been placed at the back of the *Owner/Operator Manual* in order to provide a convenient place for filing SIB's. Place the tab separator provided with this SIB in front of the title page and file this material behind the SOFTWARE INFORMATION BULLETINS tab separator in the *Owner/Operator Manual.*

DIRECTORY AND FILE MANAGEMENT SOFTWARE INFORMATION BULLETIN

INTRODUCTION

This Software Information Bulletin provides important information concerning the Directory and File Management Utilities. Please read this bulletin carefully before attempting to install or use these utilities.

The AT&T 3B2 Computer Directory and File Management Utilities can be used by anyone who has a need for enhanced file and directory manipulation. The Directory and File Management Utilities are part of the **UNIX*** System V Release 2.0 configuration provided with all 3B2 Computers.

SOFTWARE DEPENDENCIES

The Directory and File Management Utilities are dependent on the presence of the Terminal Information Utilities. The Terminal Information Utilities must be installed before the **pg** command will execute properly.

COMMAND ACCESS

There are no restrictions on these commands. Once these commands are installed, they are available to anyone who is logged in on the system.

* Trademark of AT&T Bell Laboratories

DOCUMENTATION

This Software Information Bulletin should be inserted in the *3B2 Computer Owner/Operator Manual.*

The commands provided by the Directory and File Management Utilities are described in the *3B2 Computer Directory and File Management Utilities Guide.*

RELEASE FORMAT

Storage Structure

The Directory and File Management Utilities (commands) are installed in the /**bin** and /**usr**/**bin** directories.

System Requirements

The minimum equipment configuration required for the use of the Directory and File Management Utilities is 0.5 megabytes of random access memory and a 10-megabyte hard disk.

To install the Directory and File Management Utilities software, there must be 264 free blocks of storage in the **root** file system and 966 free blocks of storage in the **usr** file system. Adequate storage space is checked automatically as part of the installation process. The installation process installs the utilities only if adequate storage space is available.

The Directory and File Management Utilities for the 3B2 Computer are distributed on one floppy disk. Most of the utilities are object code files. The **diff3** and **dircmp** commands are shell scripts.

Files Delivered

The Directory and File Management Utilities are delivered on a single floppy disk. The directory structure and files are as follows.

DIRECTORY	FILES			
/bin	ar file	od	sum	tail
/etc	magic			
/usr/bin	awk bdiff bfs comm csplit cut	diff3 dircmp egrep fgrep join newform	nl pack paste pcat pg	sdiff split tr uniq unpack
/usr/lib	diff3prog			
/usr/options	dfm.name			

UTILITIES INSTALL PROCEDURE

Use the standard software install procedure described in the *3B2 Computer Owner/Operator Manual* for the installation of the Directory and File Management Utilities.

UTILITIES REMOVE PROCEDURE

Use the standard software removal procedure described in the *3B2 Computer Owner/Operator Manual* for the removal of the Directory and File Management Utilities.

CONTENTS

Chapter 1

INTRODUCTION

PAGE

Chapter 1

INTRODUCTION

GENERAL

This guide describes command formats (syntax) and use of the Directory and File Management Utilities provided with your AT&T 3B2 Computer. The commands and procedures described in this guide can be used by anyone has a need for enhanced file and directory manipulation on the 3B2 Computer.

Directory and File Management Utilities are software tools to aid in managing your directories and files. With one-step commands, you can skillfully do any of the following:

- Search directories and files
- Compare their contents
- Manipulate file data.

GUIDE ORGANIZATION

- Chapter 2, " COMMAND DESCRIPTIONS," describes the command formats (syntax) for each command in the Directory and File Management Utilities. The descriptions include the purpose of the command, a discussion of the command syntax and options, and examples of using each command.

- Appendix, " MANUAL PAGES," contains the Directory and File Management Utilities **UNIX*** System manual pages.

* Trademark of AT&T Bell Laboratories

Chapter 2

COMMAND DESCRIPTIONS

Chapter 2

COMMAND DESCRIPTIONS

COMMAND SUMMARY

The Directory and File Management Utilities Package provided with the 3B2 Computer includes twenty-seven UNIX System commands. These commands with a brief description are listed in Figure 2-1.

Commands	**Description**
ar	Maintains groups of files that are part of a single archive file.
awk	Searches input lines for a matching pattern and performs specific actions.
bdiff	Finds which lines should be changed for two files to agree.
bfs	A read-only editor, similar to the **ed** editor, that is used to scan big files.
comm	Selects or rejects lines common to two sorted files.
csplit	Splits a file into parts as specified.

Figure 2-1. Directory and File Management Commands (Sheet 1 of 5)

Commands	Description
cut	Cuts out selected fields of data on each line of a file.
diff3	Compares three versions of a file.
dircmp	Compares two directories.
egrep	Searches a file for an *egrep* pattern, that is, a full regular expression.
fgrep	Searches a file for an *fgrep* pattern, that is, a fixed string.
file	Determines information about a file.

Figure 2-1. Directory and File Management Commands (Sheet 2 of 5)

Commands	Description
join	Joins two sorted files.
newform	Reads lines from a file or the standard input and reproduces those lines in a reformatted form on the standard output.
nl	Numbers lines in a file.
od	Outputs data in octal, decimal, ASCII, or hexadecimal formats.
pack	Stores file data in a compressed form.
paste	Merges the lines of two or more files in a side-by-side fashion.

Figure 2-1. Directory and File Management Commands (Sheet 3 of 5)

Commands	Description
pcat	Unpacks a compressed file for viewing only.
pg	Allows you to view a file a page at a time on a video display terminal.
sdiff	Compares two files to produce a side-by-side listing of different lines.
split	Splits a file into parts of equal length.
sum	Calculates the checksum and blocks of a file.

Figure 2-1. Directory and File Management Commands (Sheet 4 of 5)

Commands	Description
tail	Copies a file or portion of to standard output.
tr	Filters a file by translating specified characters to other characters.
uniq	Reports file lines that are repeated.
unpack	Stores a compressed file in uncompressed form.

Figure 2-1. Directory and File Management Commands (Sheet 5 of 5)

HOW COMMANDS ARE DESCRIBED

A common format is used to describe each of the commands. This format is as follows:

- **General:** The purpose of the command is defined. Any special or unique information about the command is also provided.
- **Command Format:** The basic command line format (syntax) is defined and the various arguments and options discussed.
- **Sample Command Use:** Example command line entries and system responses are provided to show you how to use the command.

In the command format discussions, the following symbology and conventions are used to define the command syntax.

- The basic command is shown in bold type. For example, **command** is in bold type.
- Arguments that you must supply to the command are shown in a special type. For example: **command** *argument*.
- Command options and fields that do not have to be supplied are enclosed in brackets ([]). For example: **command** [*optional arguments*].
- The pipe symbol (¦) is used to separate arguments when one of several forms of an argument can be used for a given argument field. The pipe symbol can be thought of as an exclusive OR function in this context. For example: **command** [*argument1* ¦ *argument2*]

In the sample command discussions, user inputs and 3B2 Computer response examples are shown as follows:

```
This style of type is used to show system generated
responses displayed on your screen.
```

`This style of bold type is used to show inputs entered from your keyboard that are displayed on your screen.`

```
These bracket symbols, < > identify inputs from the
keyboard that are not displayed on your screen, such
as: <CR> carriage return, <CTRL d> control d, <ESC g>
escape g, passwords, and tabs.
```

This style of italic type is used for notes that provide you with additional information.

Refer to the appendix or the *3B2 User Reference Manual* for **UNIX** System V manual pages supporting the commands described in this guide.

COMMANDS

"ar" — Archive and Library Maintainer for Portable Archives

General

The **ar** command is used to maintain groups of files that are part of a single archive file. The **ar** command is mainly used to create and update library files that are used by the link editor (**ld** command), but it can be used for any similar purpose. Refer to the *3B2 Computer Software Generation Utilities Guide* for information on the **ld** command. When the **ar** command creates an archive, the archive file is put into a format with headers that are portable across all computers that are compatible with your AT&T 3B2 Computer. These headers are placed at the beginning of each archive and have the following format.

```
#define ARMAG   "<ar>"
#define SARMAG  4


struct ar_hdr {                      /* archive header */
        char    ar_magic[SARMAG];    /* magic number */
        char    ar_name[16];         /* archive name */
        char    ar_date[4];          /* date of last archive modification */
        char    ar_syms[4];          /* number of ar_sym entries */
};
```

The header is followed by an archive symbol table which is included in each archive that has common object files. This symbol table is automatically created by the **ar** command. The archive symbol table is used by the link editor to determine which archive members must be loaded during the link edit process. There may be more than one archive symbol table. The number of symbol table entries is indicated in the header under the *ar_syms* variable. Each archive symbol table has the following format.

```
struct  ar_sym {                    /* archive symbol table entry */
        char    sym_name[8];        /* symbol name, recognized by ld */
        char    sym_ptr[4];         /* archive position of symbol */
};
```

The archive symbol table is rebuilt each time the **ar** command is used to create or update the contents of an archive.

The archive symbol table is followed by the archive file members. A file member header precedes each file member. The file member header has the following format.

```
struct  arf_hdr {                   /* archive file member header */
        char    arf_name[16];       /* file member name */
        char    arf_date[4];        /* file member date */
        char    arf_uid[4];         /* file member user identification */
        char    arf_gid[4];         /* file member group identification */
        char    arf_mode[4];        /* file member mode */
        char    arf_size[4];        /* file member size */
};
```

All the information in the archive header, the archive symbol table, and the archive file member headers is stored in a machine (computer) independent fashion. Because of this, an archive file may be used on any one of the computers that is compatible with the 3B2 Computer.

Command Format

The general format of the **ar** command is as follows.

ar *key* [*posname*] *afile name(s)*

The *key* argument uses the following options.

> ***Note:*** Options **v**, **u**, **a**, **i**, **b**, **c**, **l**, or **s** must be used in combination with at least one of options **d**, **r**, **q**, **t**, **p**, **m**, or **x**.

d Delete *name(s)* from the archive file.

r Replace *name(s)* in the archive file using one of the following options.

u	Replace only those files that have modified dates later than the archive files.
a	Place new files after *posname*.
b or **i**	Place new files before *posname*.

The *posname* argument must be specified. If you do not specify where to place new files, they will be placed at the end.

q Quickly append the named files to the end of the archive file. The positioning options under the **r** option will not work if used with this option. The **ar** command does not check to see if the added members are already in the archive.

t Print a table of contents of the archive file. If *name(s)* is specified, only that file(s) will be placed in the table of contents. If *name(s)* is not specified, all files in the archive will be placed in the table of contents.

p Print the contents of *name(s)* in the archive file.

m Move *name(s)* to the end of the archive. The positioning options under the **r** option can be used to place the file in a specific place.

x Extract *name(s)*. *If name(s) is not specified, extract all files in the archive. This option will not alter the archive file.*

v Verbose. When making a new archive from an old archive and the constituent files, a file–by–file description of the process is given. When this option is used with the **t** option, a long listing of all information about the files is given. When this option is used with the **x** option, each file is preceded by its name.

c Create *afile*. Normally, the **ar** command will create *afile* when it needs to. The normal message that is produced when *afile* is created will not appear when this option is specified.

l Place temporary files in the local directory. If this option is not specified, temporary files will be placed in **/tmp**.

s Regenerate the archive symbol table even if the **ar** command is invoked with an option that will not modify the archive contents. This option is useful to restore the archive symbol table after the **strip** command has been used on the archive. Refer to the *3B2 Computer Software Generation Utilities Guide.*

The *posname* argument is used to determine the position of a file that is being moved: either before or after *posname*. *posname* is the actual name of a file in the archive. The *posname* argument must be used when using the positioning options listed under the **r** option of the *key* argument.

The *afile* argument is the name of the archive file.

The *name(s)* argument is the name of the constituent files in the archive file. Take caution not to list *name(s)* twice. If *name(s)* is mentioned twice, it may be put in the archive twice.

Sample Commands

The following command line entries and system responses show you how to create an archive. The **ls** command is used to show you the files that will be placed in the archive. The **ar** command used with the **q** option shows you how to create an archive named **archive1**.

```
$ ls<CR>
cars
cities
people
states
streets
$ ar q archive1 cars cities people states streets<CR>
ar: creating archive1
$
```

The following command line entry and system response show you how to print a table of contents of the archive that was created in the previous example.

```
$ ar t archive1<CR>
cars
cities
people
states
streets
$
```

The following command line entry and system response show you how to print the contents of a file in an archive.

```
$ ar p archivel cars<CR>
Camero
Ferrari
Jaguar
Mustang
Porsche
$
```

The following command line entry and system response show you how to print a long listing of the table contents in an archive.

```
$ ar tv archivel<CR>
rw-------  5516/  5500    38 Jun 15 13:29 1984 cars
rw-------  5516/  5500    51 Jun 15 13:33 1984 cities
rw-------  5516/  5500    29 Jun 15 13·27 1984 people
rw-------  5516/  5500    51 Jun 15 13:48 1984 states
rw-------  5516/  5500    60 Jun 15 14:16 1984 streets
$
```

"awk" — Pattern Scanning and Processing Language

General

The **awk** command is used to search lines of input data for defined patterns and to perform specified actions on the lines or fields when a match is found. Lines that do not contain a matching pattern are ignored. Conversely, a line that contains more than one matching pattern can be operated on and output several times. One of the primary uses of **awk** is for the generation of reports. Input data is processed to extract counts, sums, and other pertinent information. The processed information is then output in a specified format.

The **awk** command has its own programming language for defining patterns and their corresponding actions. The language is designed to simplify the task of information retrieval and text manipulation. Initially, the novice user will find **awk** difficult to use and understand. Your understanding of **awk** will increase as you spend more time using (and experimenting with) the capabilities provided by the command. Remember that the use of this command is task oriented; you must establish a purpose for using the command. For example, the **awk** command can be used to output tabular material in a different sequence of columns. Certain basic arithmetic functions can also be performed on designated fields.

Input Data Characteristics

Input data is normally taken from files of data. The variable called **FILENAME** contains the name of the current input data file. Input data is divided into records with each record ended by a record separator. The record separator is stored in a variable named **RS**. The default record separator is a new-line character. This means that by default, **awk** reads and processes data on a line–by–line basis. The record separator can be redefined by setting the variable **RS** equal to the desired character. When the **RS** is empty (undefined), a blank line is used as the record separator. In addition, the field separators are defined as blanks, tabs, and new-lines. The novice user should not arbitrarily redefine the **RS** variable. The number of the current record (current line) is stored in a variable named **NR**.

Each input record (line) is divided into fields. Each field, with the exception of the last field, is ended by a field separator. The field separator is stored in a variable named **FS**. The default field separator is white space: blanks or tabs. The field separator can be redefined by setting the variable **FS** equal to the desired character. The novice user should not arbitrarily redefine the **FS** variable. Each field is identified by a unique variable. The variables are **$1**, **$2**, **$3**, and etc. The first field is named **$1**. The entire record (line) is named **$0**. The number of fields in the current record is maintained in a variable named **NF**.

Command Language Format

The instructions that tell the **awk** command what to do to the input data can be specified directly as an argument to the command, or the instructions can be read from a file. In either case, these instructions constitute an **awk** program. An **awk** program is a sequence of statements that are in the following form for each statement. Note that an action must be enclosed in braces to distinguish it from a pattern. Additional command format information is provided later in this description. The general form of each statement is as follows.

```
pattern { action }
```

The **awk** command operates on one record (line) of input data at a time. Each line of input data is tested against each of the command lines defined in an **awk** program. When a pattern match is found, the associated action is executed. When a command line defines a pattern without an associated action, then the input data record is output when a match is found. When a command line defines an action without an associated pattern, then the action is performed for all input lines (records). After all program command lines have been tested against the current record (line), the next record (line) is read and the process repeated until all records are read.

Patterns

A pattern is an expression that determines whether the associated action is to be performed. When a command line defines a pattern without an associated action, then the input data record is output when a match is found. A variety of expressions can be used as patterns. Patterns can be regular expressions as used with **ed** or **grep**, relational expressions, or special expressions. Combinations of these types of expressions can be used to define a pattern by using Boolean operators to connect each expression. The Boolean operators are OR (**||**), AND (**&&**), and NOT (**!**). Conventional arithmetic operators are also provided. The arithmetic operators are add (+), subtract (-), multiply (*****), divide (/), and modulus (**%**). Also included are the increment (++) and decrement (--) operators.

Regular Expressions: Regular expressions, in their simplist form, are context search patterns in the form used by the **ed** or **grep** commands. The **awk** language adds operators to the regular expression to specify whether the corresponding action is to be executed if the pattern matches (~) or does not match (!~). For example, the following pattern outputs all records in which the first field does not contain the word " operates" .

```
$1 !~ /operates/
```

Relational Expressions: Relationships are statements that express conditions such as greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equal to (==), and not equal to (!=). For example, the following relational expression selects lines that begin with any letter that is equal to or greater than the letter **s**. All lines beginning with the letters **s** through **z** are matched by this pattern.

```
$1 >= " s"
```

Special Expressions: Two special expressions named **BEGIN** and **END** are provided. The **BEGIN** expression defines a special pattern that matches the beginning of the input data before the first record is read. The **END** expression defines a special pattern that matches the end of the input data after the last record has been processed.

These special expressions provide the means to establish initial and post-processing conditions. When **BEGIN** is used, it must be the first pattern in the **awk** sequence of commands (program). The **END** must be the last pattern in the program. The following example shows a typical structure. In this example, the field separator (**FS**) is set to a colon at the beginning of the program; the number of each record (**NR**) is output at the end of the program.

```
BEGIN { FS = ":" }

...body of program...

END { print NR }
```

Two patterns, separated by a comma, can be used to control the execution of an action over a range of records. The action is executed for each record, starting with the match of the first pattern and continuing until the match of the second pattern, inclusive of the record containing the second pattern. The following example shows the general construction for a pattern range that controls an action.

```
/pattern1/, /pattern2/ { action }
```

Actions

An action specifies a function that is to be executed. When an action is associated with a pattern, then the action is executed only when the current record (line) matches the associated pattern. An action that does not have a corresponding pattern is executed for each input record (line). The various action terms recognized by the **awk** command are as follows.

```
exp
index(s1,s2)
int
length
log
print
printf(" f" ,e1,e2,...)
split(s,array," sep" )
sprintf(" f" ,e1,e2,...)
sqrt
substr(s,m,n)
```

Each of these action terms are described in the following paragraphs.

exp: The **exp** function computes *e* (2.7182818) raised to the *x* power, where *x* is a field argument. For example, when combined with the **print** function, the following statement outputs the value of e^x for each record, where *x* is the value of third field.

{ **print exp($3)** }

index(s1,s2): The index function is used to obtain the starting position of a string (**s2**) within another string (**s1**). A zero is returned when **s2** does not exist within **s1**. When combined with the print function, the following statement outputs the starting character position of a string **Smith** within the first field of each record.

{ **print index($1," Smith")** }

int: The integer function converts irrational numbers to rational numbers for a specified field; numbers expressed to some fractional quantity are converted to whole numbers. The function DOES NOT round off numbers. Fractional quantities are deleted. For example, the number 3.984 would be converted to the number 3. When combined with the **print** function, the following statement outputs the fifth field of each record expressed as whole numbers (integers).

```
{ print int($5) }
```

length: The **length** function computes the length of a string of characters. When combined with the **print** function, the following statement outputs the length of each record (line).

```
{ print length($0) }
```

The following statement outputs the length of each record, followed by the record.

```
{ print length($0), $0 }
```

The **length** function can also be used to output records (lines) that are within a specified length range. For example, the following statement outputs all records that are less than 20 and greater than 10 characters in length. The ¦¦ is the Boolean OR function.

```
{ length > 10 ¦¦ length < 20 }
```

The following statement outputs all records that are outside of the 10 to 20 character range.

```
{ length < 10 ¦¦ length > 20 }
```

log: The **log** function computes logarithms to the base *e*. When combined with the **print** function, the following statement outputs the logarithm of fifth field for each record.

```
{ print log($5) }
```

print: The simplist action provided is the **print** function. For example, the following statement outputs the first two fields of each record in reverse order (field 2 followed by field 1).

```
{ print $2, $1 }
```

The output of a **print** statement can be directed to a file. For example, the following statement outputs the first field to **file1** and the second field to **file3** for each record.

```
{ print $1 >> " file1" ; print $2 >> " file3"  }
```

The output of a **print** statement can be directed to another program. For example, the following statement outputs the fifth field of each record to the **sort** command. The output of the **sort** command is directed to a file named **file5**.

```
{ print $5 ¦ " sort -o file5"  }
```

printf(" f" ,e1,e2,...): The **printf** converts, formats, and prints its arguments on the standard output. This function is exactly like the C Language **printf** function. The **f** argument specifies the format. The expressions to be formatted are specified by the **e** arguments. For example, the following statement prints the third field of each record as a floating point number that is ten digits wide with two decimal places. The fifth field is printed as a ten-digit long decimal number, followed by a new-line (\n).

```
{ printf(" %8.2f %10ld\n" , $3, $5) }
```

Remember that with this print function, you must specify the output field separators; no field separators are automatically output.

split(s,array," sep"): The split function is used to automatically divide a string into fields. The **sep** argument, if provided, defines the field separator. The **FS** variable is used as the field separator if the **sep** argument is omitted from the statement. The string **s** is divided into fields defined by the **array** argument. For example, the following statement divides the second field of a record into elements of an array designated **z** based upon the dash as the field separator. Each element of the array (field) is individually identified. The first element is designated **z[1]**; the fifth element is **z[5]**. In the following example, the **print** function is used to output the fifth field of the array **z**.

```
{split($2,z," -" );print z[5]}
```

sprintf(" f" ,e1,e2,...): The string print function is used to place formatted output in a character array pointed to by a single character name. The **sprintf** converts, formats, and outputs its arguments to a string name. For example, the following statement sets **x** equal to the formatted result of the third and fifth field. The third field of each record is formatted as a floating point number that is ten digits wide with two decimal places. The fifth field is formatted as a ten-digit long decimal number followed by a new-line (\n). Thus, the variable **x** is set to the string produced by formatting the values of fields $3 and $5. The variable **x** can be used in other statements to express these values as a formatted expression.

```
{ x = sprintf(" %8.2f %10ld\n" , $3, $5) }
```

sqrt: The square root function computes the second root of a specified item. When combined with the **print** function, the following statement outputs the square root of the first field for all records.

```
{print sqrt($1)}
```

substr(s,m,n): The substring function is used to obtain a specified part of a string. The **m** argument defines the starting character position of the substring. The beginning of the string is character position number 1. The **n** argument defines the number of characters to be included in the substring. If the **n** argument is omitted, the substring is defined from the beginning position **m** to the end of the string **s**. When combined with the **print** function, the following statement outputs part of the third field for all records. The portion of the field that is selected is the fifth character position to the end of the field.

```
{ print substr($3,5) }
```

Assigning Variables

Variables are assigned as either floating point numbers or as string values. Unlike C Language, variables DO NOT have to be declared at the beginning of program. For example, the following sets **x** equal to the string **word**.

```
x = " word"
```

The following sets **x** equal to the number **100**.

```
x = " 100"
```

Arrays

Arrays are used to hold fields of data that are called elements. Each element or field in the array is identified by its sequential position. Array elements can also be named by non-numeric values, which provides an associative type of memory. For example, the following **awk** program counts the number of times the patterns **apple** and **orange** occur. The results are stored in an array designated **z**. The accumulated counts for each of these patterns is output at the end of the program. Note that the ++ operator increments the count by one each time it is called.

```
/apple/ {z[" apple" ]++}
/orange/ {z[" orange" ]++}
END {print z[" apple" ], z[" orange" ]}
```

The following example does the same function as the previous program. The only differences are that numeric designators are used for the elements of the array as opposed to associative names, and that the names are output to identify the counts.

```
/apple/ {z[1]++}
/orange/ {z[2]++}
END {print "apple = " z[1] "  orange = " z[2]}
```

Control Flow Statements

The **awk** programming language provides the following basic control flow statements: **if-else**, **while**, and **for**. Also provided are the following control statements: **break**, **continue**, and **next**. The **break** statement causes an immediate exit from an enclosing **while** or **for** construction. The **continue** statement causes the next cycle of a loop to begin. The **next** statement causes **awk** to immediately skip to the next record and begin processing.

Control flow constructions are exactly like that of the C Programming Language. For example, the following construction outputs all fields on a separate line using a **while** statement.

```
i=1
while (i<=NF) {
    print $i
    ++i
}
```

The following example construction outputs all fields on a separate line using a **for** control statement.

```
for (i=1;i<=NF;++i)
print $i
```

Commenting Programs

In general, the **awk** programs that you write are done so in files. Only the simplist of **awk** functions are accomplished by specifying patterns and actions directly to **awk** as arguments. When writing a program, the importance of adequately providing comments that indicate what you are doing at various stages in the program cannot be over emphasized.

Comments are entered by preceding the comment with a pound symbol (#). The comment ends with the end of the line. When more than one line is used for a comment, each comment line must begin with the pound symbol. Remember that if your erase character is the pound symbol, you must precede the pound with a backslash (\) to enter the symbol. This is referred to as escaping the special meaning of the character. The following shows how you enter comments into a program.

```
print x, y # Print results
# This is a continued or new comment line.
```

Command Format

The general format of the **awk** command is as follows.

awk [**-f** *source* | '*cmds*'] [*parameters*] [*file*]

The instructions that tell the **awk** command what to do can be directly expressed to the command in the form of arguments or they can be entered into a file which is then read by the command. When instructions are expressed as arguments, they are in the form *'cmds'*. Note that instructions that are expressed in the form of arguments must be enclosed in single quotes. When instructions are placed in a file, the file is specified to the **awk** command in the form **-f** *file*. Note that the file name can be expressed as a full path name.

The *parameters* argument is used to identify the value of variables. The argument is in the form of **x=... y=...**, and so on. Note that a space is used to separate each variable statement.

The *files* argument identifies the input data file. The file name can be expressed as a complete path name.

Sample Command Use

The following examples are based on a file named **list**. This file contains a list of names, addresses, and phone numbers as follows. Note that the format of each line of this file is name(tab)address(tab)phone.

```
$ cat list<CR>
Nancy                   1080 Route 3, Farmington, NC  27015           919-736-2437
John                    4589 Breckenridge, Clemmons, NC  27012        919-828-7512
Sam                     2700 Route 67, Winston-Salem, NC  27106       919-234-1940
doctor                  4100 First St, Winston-Salem, NC  27102       919-727-1111
$
```

The first example uses the **awk** command to output selected names and addresses from the **list** file in a format suitable for mailing labels. The program is in a file called **labelsprgm** and follows.

```
$ cat labelsprgm<CR>
BEGIN{FS="\t"}
$1`/Nancy/||$1~/Sam/{split($2,x,",")
printf("%s\n%s\n%s\n%s\n\n",$1,x[1],x[2],x[3])}
$
```

The second example uses the **awk** command to output selected names and phone numbers from the **list** file. The program is in a file called **numbers** and follows.

```
$ cat numbers<CR>
BEGIN{FS="\t"}
$1~/Nancy/||$1~/Sam/{printf("%s\t%s\n",$1,$3)}
$
```

The following command line entry and system responses show the use of the **labelsprgm**.

```
$ awk -f labelsprgm list<CR>
Nancy
1080 Route 3
Farmington
NC  27015

Sam
2700 Route 67
Winston-Salem
NC  27106

$
```

The following command line entry and system responses show the use of the **numbers** program.

```
$ awk -f numbers list<CR>
Nancy           919-736-2437
Sam             919-234-1940
$
```

The following example is based on a file named **gaintbl**. This file is a table containing columns of measured data, input and output voltage (**Vi** and **Vo**), and blank columns for new data.

```
$ cat gaintbl<CR>
Vi Vo Vo/Vi Log+Av Av(dB)
-- -- ----- ------ ------
 2  5
 8 15
10 18
$
```

In this example, the **awk** command performs several arithmetic operations on the data in **gaintbl**. The measured data and the resulting new data are output in a table format. Since the field separators of **gaintbl** are spaces, a **BEGIN** statement is not required. The program is in a file called **calcprgm** and follows.

```
$ cat calcprgm<CR>
#
#        PRINT TABLE HEADING
$1`/Vi/¦¦$1`/--/{
printf "%s\t%s\t%s\t%s\t%s\n",$1,$2,$3,$4,$5}
#
#      CALCULATE & PRINT DATA
$1!`/Vi/&&$1!`/--/{$3=($2/$1);$4=log($3);$5=20*$4;
printf "%21d\t%21d\t%3.3f\t%3.4f\t%3.3f\n",\
$1,$2,$3,$4,$5}
$
```

The following command line entry and system responses show the use of the **calcprgm**.

```
$ awk -f calcprgm gaintbl<CR>
Vi       Vo       Vo/Vi    Log+Av   Av(dB)
--       --       -----    ------   ------
 2        5       2.500    0.9163   18.326
 8       15       1.875    0.6286   12.572
10       18       1.800    0.5878   11.756
$
```

"bdiff" — Big Differential File Comparator

General

The **bdiff** command operates very much like the **diff** command covered later in this chapter. It compares two files and outputs instructions that tell what must be changed to bring the two files into agreement. The purpose of **bdiff** is to compare files that are too large for **diff** to process. It splits the files being compared into segments and performs **diff** on each segment. The output is identical to that of **diff**, except the line numbers are adjusted to account for the previous segments.

Command Format

The general format for the **bdiff** command is as follows:

bdiff *file1 file2* [**n**] [**-s**]

If no options are specified, **bdiff** ignores the lines which are common to the beginning of both files and splits the remainder of each file into 3500-line segments. The **diff** command is then performed automatically on the segments. The output will be the lines of the first named file followed by the lines of the second named file which are different. The less-than symbol (<) precedes the lines of the first named file. The greater-than symbol (>) precedes the lines of the second named file.

The optional third argument, **n**, is used to specify the number of lines to be contained in the file segments. If **n** is given in numeric form, the files are split into **n**-line segments instead of the 3500-line default count. These **n**–line segments are useful in those situations where the 3500-line segments are still too large for **diff** to handle.

The **-s** option will suppress any diagnostics that would be displayed by **bdiff**. However, any diagnostics output by **diff** will still be displayed.

If both the **n** and **-s** options are specified, they must be specified in the order shown in the command format, that is, the numeric value for **n** is entered before the **-s** option.

If a dash (—) is entered instead of *file1* or *file2*, the file that is named will be compared to what is input from the terminal. The input is entered exactly as it is to be compared to the named file. A "control d" is used to indicate the end of the input.

Sample Command Use

The following command line and system response shows how to output the differences between **chapter1.1** and **chapter1.2**.

```
$ bdiff chapter1.1 chapter1.2<CR>
23c23
< designed for Release 1.1 of the software.
---
> designed for Release 1.2 of the software.
104c104
< with update considerations for Release 1.2 compatibility.
---
> with update considerations for Release 1.3 compatibility.
$
```

The following command line and response demonstrates how to split the files into 1000-line segments.

```
$ bdiff file1 file2 1000<CR>
2124c2124
< is a sample of the command.
---
> is an example of how to use the command.
$
```

Note: The difference between the two files was found in the second segment, but **bdiff** adjusts the line count to specify the correct line number for the original file.

The following example shows how to format **chap1** and compare the formatted file to **OLDchap1**. The format program for this example is called **form**.

```
$ form chap1 | bdiff OLDchap1 — <CR>
72c72
< will not be displayed on the screen.
---
> will be displayed on the screen.
$
```

Note: In this example, **OLDchap1** lines will be displayed first. The order may be reversed if the order of the filename and — are reversed.

"bfs" — Big File Scanner Editor

General

> ***Note:*** This command does not follow the same format as the other commands in this Utilities Guide.

This part of the chapter describes the **bfs** (big file scanner) editor used on the 3B2 Computer. The bfs editor is similar to the *ed* editor, except that it is read-only. Since bfs cannot be used to modify a file, commands such as: insert, append, substitute, delete, and move will not execute.

Bfs works with the actual file instead of a copy placed in a buffer (temporary memory). It is normally used for processing files which are too large for conventional editing. Bfs can access files up to 1024 kilobytes (maximum possible size) and 32,000 lines--with up to 255 characters per line.

The bfs editor is useful for identifying sections of a large file where the commands *csplit* or *split* can be used to divide it into more manageable pieces for editing. The *csplit* and *split* commands are included in this Utilities Guide.

This editor description assumes that you know how to login to the 3B2 Computer. If you do not, refer to the *3B2 Computer Owner/Operator Manual.*

For additional information on the bfs editor, see the UNIX System manual pages in the Appendix.

Current Line Definition

Throughout this chapter, the term " current line" is used to identify which line in the file you are currently on. To display the current line, enter:

```
p<CR>
```

Any commands you execute will use this line as a reference point.

Getting Started

The bfs editor can only be used on existing files. To create a new file by inputting data directly, you must use another editor. However, bfs can be used to create a new file if you need to copy part of an existing file into another file. Commands to do this are discussed later in this chapter.

To execute the bfs editor, you must first be logged in to the 3B2 Computer. Once you are logged in, the UNIX System prompt ($ or #) should be displayed. You are now ready to begin working with the bfs editor.

Accessing a File

To scan a file using the bfs editor, you will need to type **bfs** followed by a space, and then the name of the file you wish to scan. Execute the command by entering a carriage return *<CR>*. For example:

```
$ bfs filename<CR>
```

will execute the bfs editor against the the file " filename" . If you entered the command correctly, the response will be a number which represents the number of characters in the edited file.

If you do not enter the command correctly, you will receive an usage message indicating an incorrect syntax was used. When this occurs, verify the name of the file; make sure you are in the right directory; and reenter the command correctly.

If you do not want the editor to display the size of the file, enter:

```
$ bfs - filename<CR>
```

where **filename** is the name of the file you want to access. The 3B2 Computer will not display a response.

Once you are in the **bfs** editor, you may begin scanning the file. To begin displaying lines in the file, you must enter a line number (for example: **1**) followed by a carriage return. The editor will use the line as a reference point. After you display a line, any of the commands described in this chapter can be used.

Displaying a Prompt

The bfs editor does not display a prompt unless you request one. At times, absence of a prompt can be confusing. Most users find it easier to use the editor with the prompt (*) displayed. To display the prompt, enter:

```
P<CR>
```

Receiving Error Messages

When the prompt is not requested, any editor error message displayed will simply be " **?**" . To receive self-explanatory error messages, the prompt must be turned on. See the previous discussion on " Displaying a Prompt."

Getting File Information

There are two editor commands which can be used to obtain information about the file you are editing. To display how many lines are in the file, enter:

```
=<CR>
```

To display the name of the file, enter:

```
f<CR>
```

Quitting the Editor

Because the **bfs** editor is read-only, it will allow you to quit without warning you to write the file. To quit the editor, enter:

> **q**<*CR*>

The 3B2 Computer will return you to the UNIX System.

Displaying Lines in the File

As previously discussed in " Current Line Definition," the current line is always displayed whenever you move through the file. However, you can display more than one line by using the print command (**p**). An example of the print command would be:

> **1,10p**<*CR*>

which would display lines 1 through 10. This form of the print command can be used to display as much of the file as you wish. The end of file symbol (**$**) can also be used with the print command to display lines. For example:

> **250,$p**<*CR*>

will display lines 250 through the end of the file. As you become familiar with the editor, you will find that the lines will be displayed even if you leave the **p** off the end of the command. For example:

> **250,$**<*CR*>

will display from line 250 to the end of the file.

The print command can also be used with other commands, such as searches and marks. These uses of the print command are discussed in the explanation of the individual commands.

Basic Movement Commands

As previously discussed, one way to move through a file is to use the carriage return. You can also use the + and - commands with the carriage return to move you forward or backward through the file. With these commands you can move to adjacent line in the file.

To move in larger steps, you can use numbers with the + and - commands. For example:

```
+15<CR>
```

will move you forward in the file 15 lines, and display the current line. Likewise,

```
-15<CR>
```

will move you backward 15 lines and then display the current line.

Each line in the file has a line number associated with it, although it is not displayed. The **bfs** editor allows you to move across large areas of the file by just entering a line number followed by a carriage return. For example:

```
375<CR>
```

will make 375 the current line and display the line.

Another movement command that is very useful on large files is the **$**. If you enter:

```
$<CR>
```

bfs will move you to the last line of the file and display the line.

Forward and Backward Searches

If you do not know a specific line number, but you do know an exact pattern of characters on a line in the file, the quickest way to locate that line is with a search. **The pattern must be on one line.** There are several types of searches. The type you should use depends on your specific application.

Searches With Wrap-Around

When entering a command, the bfs editor interprets the character " /" as meaning " search for this pattern." The search command " /" searches from the current line forward through the file for the first occurrence of the pattern. When the end of the file is reached, the search will wrap-around to the beginning of the file and continue searching until the pattern is found or it reaches the line where the search started. If the pattern is found, the line will be displayed and will become the current line. An example of a forward search command would be:

/learning the bfs editor/*<CR>*

which will search for the first occurrence of a line containing the pattern " learning the bfs editor," make it the current line, and display the line. If the pattern is not found, the message:

" learning the bfs editor not found"

will be displayed. This means the pattern you searched for is not on one line in the file, and the current line does not change. Check to see if you entered the command correctly, or if it included any characters with a special meaning (see " Special Search Characters").

The character " **?**" also executes a search when used in a command. It works the same as the " /" search character, except that it searches backward through the file from the current line. This search will wrap-around to the end of the file and continue searching until the pattern is found or it reaches the line where the search started. An example of a backward search command would be:

?learning the bfs editor?*<CR>*

Searches Without Wrap-Around

Another set of search commands can be used that do not wrap-around the end of the file. These commands are similar to the wrap-around searches, except that they stop at the beginning or end of the file. The forward search command " >" searches for the first occurrence of the specified pattern until it reaches the end of the file. An example of this type of forward search command would be:

>**learning the bfs editor**>*<CR>*

If the pattern is found, the line will be displayed and will become the current line. If the pattern is not found, the message:

" learning the bfs editor not found"

will be displayed and the current line will not change.

The character " <" also executes a search. It works the same as the " >" search, except that it searches from your current position backwards until it reaches the beginning of the file. An example of this type of backward search command would be:

<**learning the bfs editor**<*<CR>*

Repeating a Search

Quite often when searching for a pattern, the first occurrence is not the one you were actually looking for. You could repeat the entire search command, but there is a much easier way. The editor remembers the last search pattern entered. If you enter the command:

//<CR>

a forward search will look for the remembered pattern. The commands **??**, >>, and << will also repeat searches. The type of search repeated depends on the command used. The repeated search does not have to be the same type as the original search.

Global Searches

The bfs editor also allows you to perform global searches on the file. A global search is used to find all the occurrences of a specified pattern in a file. This type of search is useful when scanning for a pattern that occurs in several places. The two types of global searches that can be executed use the **g** and **v** commands.

The global search which uses the **g** command locates all the lines which contain a specified pattern. An example would be:

```
g/sample pattern/p<CR>
```

which will search for and display all lines containing the words " sample pattern." The current line will be the last line displayed.

The global search which uses the **v** command locates all lines which **do not** contain a specified pattern. An example would be:

```
v/sample pattern/p<CR>
```

which will search for and display all lines which **do not** contain the words " sample pattern." The current line will be the last line displayed.

Special Search Characters

Several characters have special meaning when used in specifying searches. These characters will work with all types of searches. They can be used to: match repetitive strings of characters, turn off special meanings of characters, or denote the placement of characters in the line. These characters are: " **.**" , " *****" , " \\" , " []" , " **$**" , and " ^" .

. The period matches any single character except the newline (carriage return) character. For example:

/bfs edit.r/<*CR*>

will search for a pattern such as " bfs editor" , " bfs editxr" , or a pattern with any other character on a line between " bfs edit" and " r" .

* The asterisk matches any string of characters except the first **.**, \\, [, or ~ in that group. For example:

/the x* editor/<*CR*>

will search for a pattern such as " the xxx editor" , " the xxxxxx editor" , or a pattern with any amount of " x" characters on a line between " the" and " editor" .

\\ The backslash is used to nullify the meaning of the special characters. It should be placed immediately before the character it is to nullify. For example:

/This is a \\$/<*CR*>

will search for the pattern " This is a $" , instead of interpreting the " $" as meaning " at end of line."

[] Brackets are used to enclose a variable string. For example:

/Search for file[23]/*<CR>*

will search for the patterns " Search for file2" or " Search for file3" and stop at the first occurrence of either pattern.

$ The dollar sign is interpreted by the editor to mean " end of the line." It is used to identify patterns which occur at the end of a line. For example:

/last character$/*<CR>*

will search for the next occurrence of a line ending in " last character" , and make it the current line.

^ The circumflex (caret) works like " **$**" except it looks for the pattern at the beginning of the line. For example:

/^**First character**/*<CR>*

will search for the next occurrence of the pattern " First character" at the beginning of a line and makes it the current line.

To search for the actual characters **.**, *****, \, [,], **$**, or ^, you must precede the characters with a backslash. This will nullify the characters special meaning.

Marking Lines

The bfs editor gives you the ability to set marks in the file. Marks are very useful when you are planning on moving around in the file and you want to set some reference points. They can save you from having to search for the same address several times.

Marks are set by moving to the line where you want the mark set and using the **k** command. The mark must be a single lowercase letter. For example, if you wanted to identify a line with the mark " a" , you would move to that line and enter the command:

ka<*CR*>

To move to that marked line from anywhere in the file, enter the command:

´a<*CR*>

The marked line will become the current line. To set another mark, repeat the **k** command using a different letter.

To change an existing mark, move to the line where you want the mark and use the **k** command with the existing mark. The new position will replace the previous one.

The **n** command will display a list of the active marks. For example, if the active marks in a file were a, b, and c, you could display them by entering:

n<*CR*>

The system response would be:

```
a
b
c
```

Notice that only the marks are displayed and not the lines.

Note: All marks are removed if you quit the editor using the **q** command. However, if you leave the editor by using the **e** command and then return to the file with the **e** command, all marks are saved. See " CHANGING FILES WHILE USING THE BFS EDITOR."

Writing to Another File

The bfs editor allows you to copy all or part of the file you are editing to another file. To copy the whole file to another file, use the **w** command and the name of the file you want to create. For example:

w newfile<*CR*>

will make a copy of the file you are editing and name it " newfile" . The number of characters in the new file will be displayed to show that the new file was created.

Caution: Be careful when naming the new file. If you use an existing filename, the text in that file will be overwritten by the new text.

If you only want to write part of the file, you must specify the beginning and ending lines you want to write. For example:

50,220w newfile<*CR*>

will make a file named " newfile" which will contain lines 50 through 220 of the file you are editing.

Changing Files While Using the "bfs" Editor

When using the bfs editor, only one file can be scanned at a time. However, the " **e**" command allows you to change files without quitting the editor. For example, if you are scanning *file1* with the bfs editor and want to change to *file2*, you would use the command:

e file2<*CR*>

This will cause the editor to leave *file1* and enter *file2*. To reenter *file1*, you would need to use the **e** command again. Using the quit (**q**) command will cause you to leave the file you are presently in and return you to the UNIX System.

Issuing UNIX System Commands

The bfs editor allows you to execute a single UNIX System command by entering a command of the form:

```
!cmd<CR>
```

where " cmd" represents the command you want to execute. The system will then execute the command. When finished, bfs will display an **!** and then return you to the current line in the file. You can then continue editing or issue another **!** command.

If you need to execute more than one UNIX System command, enter the command:

```
!sh<CR>
```

When you are finished executing UNIX System commands, enter a **control-d** (depress and hold the CONTROL " CTRL" key and simultaneously depress the " d" key). The editor will display an **!** and return to the current line in the file.

High-Level "bfs" Commands

A " command file" is an executable file that contains editor commands. Command files may be set up and run against other files with the bfs editor. When executing command files, the output is directed to another file. Information on the other commands which can be used in command files is given in the Appendix.

"comm" — Select or Reject Common Lines

General

The **comm** command compares two files and produces an output showing the differences and similarities between them. The contents of the two files should be in alphabetical order, that is, in order according to the ASCII collating sequence. The output is formatted into three columns. The first column lists those lines found only in the first named file, the second column lists those lines found only in the second named file, and the third column lists those lines that are common to both files.

Command Format

The general format for the **comm** command is as follows:

comm [**—** [**123**]] *file1 file2*

The **—** [**123**] option suppresses the column corresponding to the number specified. For example, if **—1** is specified, the first column of the output is not displayed. Thus, only those lines unique to the second named file and those lines common to both named files are displayed.

If a dash (—) is entered in place of a file name, the standard input from the terminal is read. The **comm** command compares the input with the file and produces the three-column output as before.

Sample Command Use

The examples provided are based on the contents of two files named **listA** and **listB**. The contents of these two files are as follows.

listA:	**birds**	**listB:**	**birds**
	cats		**cats**
	dogs		**horses**
	horses		**mice**
	mice		**mules**
	snakes		**pigs**

The following command line and system response shows how to compare the two lists and receive all columns of the output.

```
$ comm listA listB<CR>
                birds
                cats
dogs
                horses
                mice
        mules
        pigs
snakes
$
```

The following command line and system response shows how to compare the two lists and output only those lines that are common to both files.

```
$ comm -12 listA listB<CR>
birds
cats
horses
mice
$
```

The following example shows how to alphabetize a file named **list** and compare it to **listB** with the same command line. The file **list** is:

birds
mice
dogs
cats
pigs

The command line uses **sort** to alphabetize the file **list**.

```
$ sort list ¦ comm - listB<CR>
                birds
                cats
dogs
        horses
                mice
        mules
                pigs
$
```

Note: The **sort** command is explained in the *UNIX System V User Guide*.

"csplit" — Context Split

General

The **csplit** command splits a file into sections using input arguments as the boundaries of the sections. The sections are suffixed with a number starting with 00 and may go up to 99. The first section (00) will contain from the beginning of the file up to, but not including, the line defined by the first argument. The second section (01) will contain the line defined by the first argument up to, but not including, the line defined by the second argument. The last section will contain the line defined by the last argument through the end of the original file. The original file is not affected.

Command Format

The general format for the **csplit** command is as follows:

csplit [**-s**] [**-k**] [**-f** *prefix*] **filename arg1** [**arg2 ... arg***n*]

The **csplit** command normally outputs the character count of each section as the section is created. The **-s** option will suppress the printing of these character counts. The process is complete when the system response is returned.

If an error occurs during the **csplit** operation, the sections that have been created are removed. The **-k** option overrides the removal of previously created files. However, the process will halt at the point the error occurred. The current section and the remainder of the original file will not be processed.

The created files are normally named **xx00** thru **xxnn**. If the **-f** *prefix* option is used, the files are named *prefix***00** thru *prefix***nn**.

The **filename** is the name of the original file that is to be split. The command will start at the beginning of the file and search for the first argument. That section is then written into a file, and the argument is used as the beginning for the next section. The arguments for the **csplit** command can be any combination of the following:

/string/	A file will be created from the current line up to, but not including, the line containing the character string *string*. This string may be followed by a +*n* or -*n* where *n* is a number of lines. For instance, if your file should contain **Page 5** and the three lines that follow it in the original file, the expression would be **/Page 5/+3**. If the character string has blanks or other meaningful characters to the command, the string must be enclosed in quotes.
%string%	This argument acts exactly like */string/* except that no file will be created for the section from the current line to the line containing *%string%*.
zzz	A file will be created from the current line up to, but not including, line number *zzz*. The line numbered *zzz* would then become the current line, that is, the first line in the next section.
{*num*}	The argument that appears before {*num*} will be repeated *num* times. If the argument is a *string* type argument, that argument is searched for *num* more times. By using {*num*} after the *zzz* argument, you can split a file *num* times every *zzz* lines. It is a good idea to use the **-k** option with this argument because if the {*num*} number is too high, you will receive an error message and lose the files that have already been created.

Sample Command Use

The following example shows how to split the file **basic** into three pieces, **bas00, bas01**, and **bas02. The first line of bas01** will contain the string *test procedures.* The first line of **bas02** will contain the string *2.05.*

```
$ csplit -f bas basic "/test procedures/"  /2.05/<CR>
2345
1068
297
$
```

The following example shows how to split the file **doc** into pieces of 100 lines each. To be sure that the entire file is split, an arbitrary number of 99 has been used for the number of times to split the file. Any lines over 10,000 will not be split. The **-s** option is used to suppress the character counts of each 100-line file.

```
$ csplit -s -k doc 100 {99}<CR>
$
```

The 100-line files would be named **xx00** thru **xx***nn*.

The following example shows how to save the last piece of the file **mail**. The saved file, **xx00**, contains the text from the line **MISC** to the end of the file.

```
$ csplit mail %MISC%<CR>
$
```

"cut" — Output Selected Fields of a File

General

The **cut** command is used to output selected columns or fields from a line of data. The lines of data operated on by the **cut** command can be from one or more files, the output of another command, or from the terminal (standard input).

Command Format

The general format of the **cut** command is as follows.

cut -c*list* [*file(s)*]

cut -f*list* [**-d***char*] [**-s**] [*file(s)*]

The **-c***list* argument identifies the character positions in each line that are to be output. Individual character positions are identified by integers. A comma (,) is used to separate each position identifier. Ranges are specified by using a dash between the starting and ending number in the range. For example, character positions 1, 5, and 7 through 10 are identified as follows.

-c1,5,7-10

The **-f***list* argument identifies the field positions of each line that are to be output. A comma (,) is used to separate each field identifier. Ranges are specified by using a dash between the starting and ending number in the range. For example fields 1, 5, and 7 through 10 are input as follows.

-f1,5,7-10

The **-s** option is used with the **-f***list* argument to prevent lines that do not contain field delimiters from being output.

The **-d***char* argument identifies the field delimiter. The default field delimiter is the tab character. For example, the argument **-d:** defines a colon as the field delimiter. Delimiter characters that have a special meaning to the shell must be either placed in single quotes or escaped by preceding the character with a backslash (\). For example, the space can be defined as a field delimiter by the following.

-d' '

The *file(s)* argument identifies the name or names of the files that are to be operated on by the command.

Sample Command Use

The following sample command line entries and system responses show you how to output the character positions 5 through 10 and 15 from each line of a file named **list**. In this example, the **cat** command is first used to display the contents of the **list** file.

```
$ cat list<CR>
ABCDEFGHIJKLMNOPQRSTUVWXYZ
         11111111112222222
12345678901234567890123456
$ cut -c5-10,15 list<CR>
EFGHIJO
     11
5678905
$
```

The following sample command line entries and system responses show you how to output the second and fifth fields from a file named **table**. The field separator (delimiter) is a colon (:). Note what happens when the delimiter is defined as a space by the **-d' '** argument. Also, note that the sequence in which you define the fields (**-f5,2** vs **-f2,5**) does not change the sequence in which they are output. The selected fields are output in the order that they appear in the data, from left to right. In this example, the **cat** command is used to display the contents of the **table** file.

```
$ cat table<CR>
field 1:field 2:field 3:field 4:field 5:field 6
field 1:field 2:field 3:field 4:field 5:field 6
field 1:field 2:field 3:field 4:field 5:field 6
$ cut -f2,5 -d: table<CR>
field 2:field 5
field 2:field 5
field 2:field 5
$ cut -f5,2 -d':' table<CR>
field 2:field 5
field 2:field 5
field 2:field 5
$ cut -f2,5 -d' ' table<CR>
1:field 4:field
1:field 4:field
1:field 4:field
$
```

"diff3" — 3-Way Differential File Comparator

General

The **diff3** command compares three files and outputs information showing the range of lines which differ between the files. The information is separated by a string of equal signs (====) to signify which files are different. If the string of equal signs is alone, this indicates that all files differ. If the string of equal signs is followed by a number, the number signifies which file is different. For example, ====**2** would indicate that the second named file is different and the information following the ====**2** would show the differences.

The range of lines which are different are shown in the format " **f:ln ed**" where;

f = the number of the file as it was entered in the command line.

ln = the line number of the line which is different. This could be a range of lines.

ed = the editor command which needs to be performed to bring the files into agreement with each other. If the **c** (change) operation is indicated, the original contents of the file will be shown immediately after the range of lines information.

Command Format

The general format for the **diff3** command is as follows:

diff3 [**-ex3**] *file1 file2 file3*

The **-e** or **-x** options publish a script file with the editor commands needed to make the first named file agree with the third named file. The script file contains all the commands necessary to make the proper changes and may be applied directly to the file.

Sample Command Use

The sample commands used in this section are based on the usage of three files named **a**, **b**, and **c**. The contents of the three files are shown below:

a:	**A**	**b:**	**B**	**c:**	**C**
	B		**C**		**D**
	C		**D**		**E**
	D		**E**		**A**
	E		**A**		**B**

The following command and system response show how to compare the three files and have the standard output displayed.

```
$ diff3 a b c<CR>
====
1:1,2c
  A
  B
2:1c
  B
3:0a
====
1:5a
2:5c
  A
3:4,5c
  A
  B
$
```

The following command line and system response show how to compare the three files and receive an editor script file which will make **a** agree with **c**.

```
$ diff3 -e a b c<CR>
5a
A
B
1,2c
w
q
$
```

The script file that is produced can be applied directly to the file being changed. This can be done on the same command line as the **diff3** command. The following command line shows how to compare the three files and apply the script file to **a**. There will be no output from this command, completion is indicated with the system prompt.

```
$ diff3 -e a b c ¦ ed - a<CR>
$
```

"dircmp" — Directory Comparison

General

The **dircmp** command compares two directories and outputs information about the names and contents of the files in each directory. The output is paginated to list those files which are unique to each directory and then those files which have common file names. The files common to both directories are compared, and the output includes whether the contents are the " same" or " different."

Command Format

The general format for the **dircmp** command is as follows:

dircmp [**-d**] [**-s**] *dir1 dir2*

The **-d** option makes a comparison of the files common to both directories and gives information on what must be done to bring the two files into agreement. The format for the output is identical to the format of the **diff** command covered previously in this chapter.

The **-s** option will suppress any messages about identical files. That is, the output will only contain information on the files that are different from each other.

Sample Command Use

The examples provided in this section are based on the contents of the two directories **dir1** and **dir2**. The contents of the two directories follow.

dir1:	**appendix**	**dir2:**	**appendixA**
	chap1		**appendixB**
	chap2		**chap1**
	chap3		**chap2**
	chap4		**chap3**
	index		**chap4**
	table		**index**
	toc		**toc**
	trademarks		**trademarks**

The following command line and system response show how to compare **dir1** and **dir2**.

```
$ dircmp dir1 dir2<CR>

May 15 09:02 1984   ../dir1 only and ../dir2 only Page 1

./appendix                                               ./appendixA
./table                                                  ./appendixB




May 15 09:02 1984  Comparison of ../dir1 ../dir2 Page 1

directory        .
same             ./chap1
different        ./chap2
different        ./chap3
different        ./chap4
same             ./index
different        ./toc
same             ./trademarks



$
```

Note: The output used in this example contains only the text of the actual output. The actual output is paginated with " white space" separating the unique files in each directory from the section displaying the common file names.

The following command line shows how to compare the files in **dir1** and **dir2** and output information that tells what must be done to bring the files into agreement.

```
$ dircmp -d dir1 dir2<CR>

 (The first part of the output would appear
 the same as in the previous example. The
 last part of the output would be in the
 format identical to that of the diff
 command.)

$
```

"egrep," "fgrep" — Search a File for a Pattern

General

The **egrep** and **fgrep** commands search files or input lines for matching character patterns. These commands are very similar to the **grep** command explained in the *UNIX System V User Guide.*

The input data to be searched can be the output of another command, one or more specified files, or the input from the terminal. When more than one file is searched, the file name is printed along with the matching input lines. The character patterns are regular expressions or fixed strings of characters in the style of the text editor (**ed**). Be careful when using the characters that have special meaning to the editor shell. In general, the pattern should be enclosed in single quotes ('pattern') to remove any special character meaning.

The *expression* **grep** (**egrep**) searches for full regular expressions. The **egrep** command accepts the following conventions for defining expressions.

- A pattern followed by a plus sign (+) matches one or more occurrences of the pattern.

- A pattern followed by a question mark (?) matches 0 or 1 occurrences of the pattern.

- Multiple patterns can be defined by separating each pattern by a pipe symbol (|) or by a new-line (carriage return). When a new-line is used, the secondary system prompt (>) is displayed. Each pattern is entered on a separate line following the prompt. The last pattern is entered on the same line as the remainder of the command. The command outputs matches for any or all patterns.

- Patterns can be grouped by enclosing the pattern in parentheses.

The *fast* **grep** (**fgrep**) command searches for fixed patterns. This command is fast and compact.

Command Format

The general format for each of these commands is as follows:

egrep [*options*] [*expression*] [*file(s)*]

fgrep [*options*] [*string(s)*] [*file(s)*]

The *options* recognized by these commands are explained as follows:

-b	Outputs the block number of the matching line. Each line is preceded by the number of the data block containing the line.
-c	Outputs only the number of lines that match the pattern.
-e *expression*	Same as a simple *expression* argument, but is useful when the *expression* contains a —.
-f *file*	The *pattern* (expressions or strings) are read (taken) from the specified *file*.
-l	Outputs only the names of the files that contain matching lines.
-n	Each line is preceded by its relative line number in the file.
-v	Outputs the lines that DO NOT contain the defined pattern.
-x	Outputs the lines that match the pattern exactly and entirely. This option is used with the **fgrep** command only.

The *expression* and *string* arguments define the search pattern or patterns. The *file(s)* argument is used to identify the file or files that are to be searched. Note that the file names are separated by a space.

Sample Command Use

The following command line and system response show how you can search two files (**list1** and **list2**) for lines containing one of several patterns. The patterns to be searched for are *eggs* and *bacon*. The **-n** option is used to display the line number of the matching line.

```
$ egrep -n 'eggs¦bacon' list1 list2<CR>
list1:2:eggs
list2:1:bacon
list2:3:eggs
$
```

Note: The semicolon (:) is used to separate each field of the output of the **egrep** command. The first field is the file name. The second field is the line number of the matched pattern in the named file. The last field is the line containing the matching pattern.

The following command line and system response show you how to enter the previous example using a new-line (carriage return) to separate the patterns instead of a pipe symbol (¦).

```
$ egrep -n 'eggs<CR>
> bacon' list1 list2<CR>
list1:2:eggs
list2:1:bacon
list2:3:eggs
$
```

The following command line and system response show how you can search multiple files for lines that DO NOT contain one of the specified patterns. The **-v** option causes all lines that DO NOT match the specified patterns to be output. The **-n** option is used again to output the line number of the matching line.

```
$ egrep -nv 'eggs¦bacon' list1 list2<CR>
list1:1:milk
list1:3:toast
list1:4:ham
list2:2:milk
list2:4:juice
list2:5:bread
$
```

The following example shows how you can use a file containing a list of patterns to search a group of files. The file to search from is named **words** and contains the following patterns: **570ab[3-7], 448hj2, 747bg32**. The first line of the **words** file defines a pattern beginning with 570ab and ending with 3, 4, 5, 6, or 7. The **egrep** command will search all files in the current directory that begin with the characters **serials**.

```
$ egrep -f words serials*<CR>
serialsnet:570ab4
serialsnet:570ab5
serialsold:747bg32
serialsnew:570ab3
serialsnew:570ab7
serialsnew:448hj2
$
```

"file" — Determine File Type

General

The **file** command is used to determine the contents of one or more specified files. The command examines the contents of the first block of data (1024 bytes) of each file and attempts to classify the data. A file called **/etc/magic** is used by the **file** command to classify files containing certain special numeric or string constants. If you **cat /etc/magic**, an explanation of the **magic** file format is displayed.

Some of the file-types that can be classified are:

3b2/3b5 executable
3b2/3b5 executable not stripped
ascii text
c program text
commands text
data
directory
empty
English text
[nt]roff, tbl, or eqn input text

Note: The file must have read permission before a classification can be made.

Command Format

The general format of the **file** command is as follows.

file [**-c**] [**-f** *ffile*] [**-m** *mfile*] *name(s)*

The **-c** argument causes the command to check the magic file for format errors. No file classification is done with this function.

The **-f** *ffile* option is used to specify the name of a file that contains a list of file names that are to be examined. The *ffile* argument identifies the name of the file containing the list of file names to be examined.

The **-m** *mfile* option is used to specify an alternate magic file. The *mfile* argument identifies the name of an alternate magic file.

Sample Command Use

The following command line entry and system responses show how you can determine the classification of a given file.

```
$ file /f1/house/bills/electric<CR>
/f1/house/bills/electric:       ascii text
$
```

The following command line entries and system responses show how you can determine the classification of several files. A file named **list** is first created that contains a listing of the files to be examined. The **file** command is then executed with the **-f** option to classify the files identified in the **list** file.

```
$ ed list<CR>
?list
a<CR>
/f1/house/bills/electric<CR>
/f1/house/bills/water<CR>
/f1/house/bills/gas<CR>
/f1/house/bills/telephone<CR>
.<CR>
w<CR>
94
q<CR>
$ file -f list<CR>
/f1/house/bills/electric:       ascii text
/f1/house/bills/water: ascii text
/f1/house/bills/gas:   ascii text
/f1/house/bills/telephone:      ascii text
$
```

"find" — Find Files

General

The **find** command is used to look for files that match certain conditions and then perform a specified function(s) when a match is found. The **find** command descends a directory structure for each path name identified to the command.

Some of the common uses of the find command include moving directories (and their contents), and removing certain files as part of general file system housekeeping. These uses of the **find** command require the use of other commands in conjunction with the **find** command to copy or remove files of data.

Command Format

The general format of the **find** command is as follows.

find *path-name-list expression*

The *path-name-list* argument identifies the directory path names that are to be searched. The *expression* argument is made from the following terms, depending on what function(s) are to be performed by the **find** command. Note that the *n* argument used with these terms is expressed as a positive, negative, or unsigned decimal number. A positive number means more than *n*. A negative number means less than *n*. An unsigned number means exactly *n*.

-name *file*	True when a file name found in the specified directory path matches the specified *file* name.
-perm *onum*	True when the access permissions match the specified octal number (onum).
-type *c*	True when the file type identifier matches the *c* argument. The file type identifiers

	are **b** (block), **c** (character special), **d** (directory), **p** (pipe), and **f** (plain).
-links *n*	True if the file has *n* links.
-user *uname*	True when the owner of the file is *uname*.
-group *gname*	True when the group identifier is *gname*.
-size *n*[**c**]	True when the file is allocated *n* blocks. A block is 512 bytes. If *n* is followed by a **c**, the size is in characters.
-atime *n*	True when the file has been accessed in *n* days.
-mtime *n*	True when the file has been modified in *n* days.
-ctime *n*	True when the file has been modified in *n* days.
-exec *cmd*	True when the executed *cmd* returns an exit code of zero. The end of the *cmd* must be an escaped semicolon (\;). The command argument {} is replaced by the current path name.
-ok *cmd*	This term functions like the **-exec** term with the addition of a prompt that asks permission to execute the command. The generated command line is output with a question mark at the end of the command line. The command is executed if you respond with a **y**<*CR*>. Any other response continues the **find** command without executing the generated command line.
-print	Causes the current path name to be output.

-cpio *device*	Write the current file on *device* in format of a cpio archive.
-newer *file*	True when the current file has been modified more recently that the *file* argument.
\\(*expression* \\)	True when the *expression* in parentheses is true. Note that the parentheses must be preceded by a backslash (escaped).
-depth	Causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This term is helpful when the **find** command is used with the **cpio** command to transfer files that are contained in directories without write permission. Refer to the *UNIX System V User Guide* for more information about the **cpio** command.

Any of these terms can be combined with one another by using the following operators.

!	The NOT operator.
-o	The OR operator.

Note that the AND operator is implied by stating terms in succession.

Sample Command Use

The following command line entries and system responses show how you can search for a specific file name starting at the current directory. All branches under the current directory are searched for the specified file. This example is searching for a file named **electric**. When a matching file name is found, the path name from the starting search point to the file is output.

Two forms of the **find** command are shown. The first form uses the " dot" file to identify the starting point of the search. The second form uses the current directory path name to identify the starting point. Each of these commands accomplishes the same thing; but, produces different forms of path names.

```
$ find . -name 'electric' -print<CR>
./bills/electric
$ pwd<CR>
/f1/house
$ find /f1/house -name 'electric' -print<CR>
/f1/house/bills/electric
$
```

The following command line entries and system responses show how you use the **find** and **cpio** commands to copy entire directories to another directory structure. The options provided with the **cpio** command copy data to a directory structure (**-p**), create directories under the new structure as needed (**-d**), and retain the previous file modification times (**-m**). The new destination directory is first created using the **mkdir** command. The working directory is then changed to the source directory using the **cd** command. The source directory named **bills** and everything under that directory is then copied to a new directory structure. After the directory structure has been copied, the total number of blocks of data copied by the **cpio** command are reported. The **cd** and **ls -l** commands are used to show the results.

```
$ cd /f2/fred<CR>
$ mkdir bills<CR>
$ cd /f1/house/bills<CR>
$ find . -print ¦ cpio -pdm /f2/joan<CR>
7 blocks
$ cd /f2/fred<CR>
$ ls -l /f2/fred<CR>
total 1
drwxrwxrwx   2 fred     plangrp     96 Jun 22 11:08 bills
$ ls -l /f2/joan/bills<CR>
total 7
-rw-rw----   1 fred     plangrp    439 Jun  7 01:06 electric
-rw-rw----   1 fred     plangrp    681 Jun  7 01:07 gas
-rw-rw----   1 fred     plangrp    241 Jun  7 01:05 telephone
-rw-rw----   1 fred     plangrp   1364 Jun  7 01:08 water
$
```

The following command line entries and system responses show how you can search under your login directory for all files that have not been accessed for over 30 days and list the files. The options used with the **ls** command omit the owner and group names from the long list format. The same command line with a **rm** command in place of the **ls -og** command can be used to remove all files that have not been accessed in over 30 days.

```
$ find $HOME -atime +30 -exec ls -og {} \;<CR>
-rw-rw----   1  2336 May 31 05:24 /f1/fred/mbox
-rw-rw----   1  2006 Mar 29 04:24 /f1/fred/car/gas
-rw-rw----   1  3200 Apr 23 03:52 /f1/fred/car/loan
-rw-rw----   1    59 Jan 29  1982 /f1/fred/house/list
$
```

"join" — Relational Database Operator

General

The **join** command is used to join a common field of two files. The results are printed on your terminal screen. The fields that are to be joined must be sorted in an increasing ASCII collating sequence. Normally, the first field in each line is the field to be joined. A blank, tab, or new-line usually separates the fields. Multiple separators will be counted as one, and leading separators are discarded.

One line of output is generated for each pair of lines in the files that have identical join fields. The output line normally consists of the common field followed by the rest of the line from the first file, followed by the rest of the line from the second file.

Command Format

The general format of the **join** command is as follows.

join [*options*] *file1 file2*

The following options exist:

-a*n*	In addition to the normal output, a line is produced for each unpairable line in file *n* (where n is 1 or 2).
-e *s*	Replace empty output fields by string *s*.
-j*n m*	Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.
-o *list*	Each output line comprises the fields specified in *list* and each element of *list* has the form *n.m* (where *n* is a file number and *m* is a field number).
-t*c*	Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

The *file1* and *file2* arguments are the names of the files that are to be joined.

Sample Command Use

The following command line entries and system responses show the basic operation of the **join** command. The **cat** command is used to display the contents of **file1** and **file2**. The **join** command is used to join an inventory of red and blue items.

```
$ cat file1<CR>
1. red balls (7)
2. red bicycles (3)
3. red cars (5)
4. red ink pens (10)
5. red shoes (2 pair)
$ cat file2<CR>
1. blue balls (4)
2. blue bicycles (2)
3. blue cars (3)
4. blue ink pens (5)
5. blue shoes (3 pair)
6. blue pants (2 pair)
$ join -a2 file1 file2<CR>
1. red balls (7) blue balls (4)
2. red bicycles (3) blue bicycles (2)
3. red cars (5) blue cars (3)
4. red ink pens (10) blue ink pens (5)
5. red shoes (2 pair) blue shoes (3 pair)
6. blue pants (2 pair)
$
```

"newform" — Change the Format of a Text File

General

The **newform** command is used to read lines from a file or the standard input, reformat those lines, and reproduce the lines on the standard output. The format is selected through the command line options listed under " Command Format."

Command Format

The general format of the **newform** command is as follows.

> **newform** [**-s**] [**-i***tabspec*] [**-o***tabspec*] [**-b***n*] [**-e***n*] [**-p***n*] [**-a***n*]
> [**-f**] [**-c***char*] [**-l***n*] [*file(s)*]

where:

-i*tabspec*	This option expands tabs into spaces. The *tabspec* part of this option uses the same tab specifications used with the **tab** command (refer to the *UNIX System V User Guide*). Tab specifications may be found on the first line of the standard input. In this case, use a double minus sign (**--**)as the *tabspec*. If *tabspec* is not specified, **-8** is used. If **-0** is given as the *tabspec*, there should not be any tabs in the text. If tabs are found in the text, they are treated as **-1**.
-o*tabspec*	This option will replace spaces with tabs. The *tabspec* part of this option uses the same tab specifications as the *tabspec* part of the **-i***tabspec* option. If *tabspec* is not specified, **-8** is used. If a *tabspec* of **-0** is specified, spaces will not be converted to tabs.
-l*n*	The effective line length is set to *n* characters. If this option is not specified, the effective line length is 80 characters. If **-l** is specified without

n, the effective line length is set to **72**. Tabs and backspaces are considered to be one character. Remember, tabs may be expanded to spaces by the **i***tabspec* option.

-b*n*	Shorten the beginning of the line by *n* characters when the line length is greater than the effective line length set by the **-l***n* option. If *n* is not specified, the line will be shortened by the amount of characters necessary to obtain the effective line length set by the **-l***n* option. It is a good idea to specify **-l***n* as **-l1** when using this option. That way, you will be sure that this option will be activated, because the effective line length will be shorter than any line in the file.
-e*n*	This option works the same as the **-b***n* option except that characters are removed from the end of the line.
-c*k*	Change the prefix character (see **-p***n* option) and/or the append character (see **-a***n* option) to *k*.
-p*n*	Prefix *n* characters to the beginning of a line when the line length is less than the effective line length set by the **-l***n* option. Spaces will be the prefix if the **-c***k* option is not specified. If *n* is not specified, the number of characters necessary to obtain the effective line length will be the prefix number.
-a*n*	This option is the same as the **-p***n* option except that characters are appended to the end of a line.
-f	Write the tab specification format line on the standard output before printing the output. The **-o***tabspec* option determines what tab specification format line is printed. If the *tabspec* part of the **-o***tabspec* option is not

	specified, the line printed will be the default specification of **-8**.
-s	Shears the leading characters off of each line up to the first tab. Up to eight of the sheared characters are placed at the end of the line. If more than eight characters are sheared, the eighth character is replaced by an * and the rest are discarded. The first tab is always discarded. There must be a tab on each line of the file. If there is not a tab on each line, an error message and a program exit will occur. The characters sheared off are saved internally until all other options specified are applied to that line. These characters are then added at the end of the processed line.
files(s)	The name of the file(s) that is to be read.

The command line options may appear in any order, may be repeated, and may be intermingled with *file(s)*. However, if you use the **-s** option, it must be the first option specified.

Sample Command Use

The following command line entries and system responses show you a typical **newform** command output. The **cat** command is used to display the contents of **testfile**. The **newform** command is used to display the contents of **testfile** while removing the first three characters of each line and keeping the same column definition.

```
$ cat testfile<CR>
   RENTAL ITEM  DATE RENTED      DATE RETURNED

1. ladder       6/5/84           6/6/84
2. lawn mower   6/6/84           6/7/84
3. spray gun    6/7/84           6/8/84
4. tiller       6/8/84           6/9/84
5. weed eater   6/9/84           6/10/84
$ newform -i -l1 -b3 testfile<CR>
RENTAL ITEM  DATE RENTED      DATE RETURNED

ladder       6/5/84           6/6/84
lawn mower   6/6/84           6/7/84
spray gun    6/7/84           6/8/84
tiller       6/8/84           6/9/84
weed eater   6/9/84           6/10/84
$
```

The following command line entry and system response show how to display the contents of **testfile** without the last column.

```
$ newform -i -l1 -e13 testfile<CR>
   RENTAL ITEM  DATE RENTED

1. ladder       6/5/84
2. lawn mower   6/6/84
3. spray gun    6/7/84
4. tiller       6/8/84
5. weed eater   6/9/84
$
```

"nl" — Line Numbering Filter

General

The **nl** command is used to read lines from a file or the standard input. The lines that are read are numbered and printed on your terminal screen. The way the lines are numbered depends on the options you select.

The **nl** command views the text it reads in terms of logical pages. A logical page contains three sections: a header, a body, and a footer section. You can have empty sections. The options for numbering lines can be different for each of the three sections. In order for the three sections to be recognized, the following delimiter character(s) must be included in the input lines.

LINE CONTENTS	START OF
\:\:\:	header
\:\:	body
\:	footer

Note: There must not be any other input on the lines containing the delimiter character(s).

The **nl** command assumes the text being read is a single, logical page body unless you select other options.

Command Format

The general format of the **nl** command is as follows.

nl [**-h***type*] [**-b***type*] [**-f***type*] [**-v***start#*] [**-i***incr*] [**-p**] [**-l***num*] [**-s***sep*] [**-w***width*] [**-n***format*] [**-d***delim*] *file* where:

-h*type*	Used to specify which logical page header lines are to be numbered. The recognized types are: **a** — Number all lines **t** — Number lines with printable text only **n** — No line numbering **p***string* — Number only the lines that contain the regular expression specified in *string*. The default *type* for the logical page header is **n**.
-b*type*	Used to specify which logical page body lines are to be numbered. The recognized types are the same as **-h***type*. The default *type* for the logical page body is **t**.
-f*type*	Used to specify which logical page footer lines are to be numbered. The recognized types are the same as **-h***type*. The default *type* for the logical page footer is **n**.
-v*start#*	The initial value used to number the logical page lines. Default is **1**.
-i*incr*	The increment value used to number the logical page lines. Default is **1**.
-p	Do not restart numbering at the logical page delimiters.
-l*num*	The number of blank lines to be considered as one. The appropriate **-ha, -ba**, and **-fa**

	option must be set. A **-l2** results in only the second adjacent blank line being numbered. Default is **1**.
-s*sep*	The character(s) used in separating the line number and the corresponding text line. Default is a tab.
-w*width*	The number of characters to be used for the line number. Default is **6**.
-n*format*	The line numbering format. The recognized values are:

ln	Left justified with no leading zeros
rn	Right justified with no leading zeros
rz	Right justified with leading zeros.

Default is **rn**.

-d*delim*	Used to change the delimiter characters. If only one character is entered, the second character remains the default character (:). If you wish to use a backslash (\\) as a delimiter character, you need to enter two backslashes (\\\\).
file	The name of the file that is read by the **nl** command.

The options can be specified in any order and can be intermingled with an optional file name. However, when intermingling options with a file, only one file can be moved.

Sample Command Use

The following command line entries and system responses show the basic operation of the **nl** command. The **cat** command is used to show the contents of file1. The **nl** command is used with several options to show how the command is inputted and the system response.

```
$ cat file1<CR>
\:\:\:
THIS IS THE HEADER SECTION
...
\:\:
This is the body section.
...
...
...
...
\:
THIS IS THE FOOTER SECTION
...
...
$ nl -ha -fa -v5 -i5 -nrz file1<CR>

000005          THIS IS THE HEADER SECTION
000010          ...

000015          This is the body section.
000020          ...
000025          ...
000030          ...
000035          ...

000040          THIS IS THE FOOTER SECTION
000045          ...
000050          ...
$
```

"od" — Octal Dump

General

The **od** command is used to output data in octal, decimal, ASCII, or hexadecimal formats. The name of the command, octal dump, is derived from the default output. Input can be from a named file, the output of another command, or from the standard input.

Command Format

The general format of the **od** command is as follows.

od [**-bcdoscx**] [*file*] [[+]*offset*[**.**][**b**]]

The meaning of the various format options are as follows.

-b	Interpret bytes in octal (base 8).
-c	Interpret bytes in ASCII.
-d	Interpret words in unsigned decimal (absolute value).
-o	Interpret words in octal. This is the default when no option argument is supplied.
-s	Interpret 16-bit words in signed decimal.
-x	Interpret words in hexadecimal (base 16).

The *file* argument identifies the name of the file to be output. If the *file* argument is omitted, input is taken from the standard input.

The *offset* argument identifies where the output is to start. This argument is normally expressed as the number of octal bytes to be skipped before data is output. If a period (.) is appended to the *offset* argument, the argument is interpreted as a decimal number of bytes. If a letter **b** is appended to the argument, the argument is interpreted as the number of blocks to be skipped before data is output. If the *file* argument is omitted, the *offset* argument must be preceded by a plus sign (+) to identify what follows as being the *offset* argument.

Sample Command Use

The following command line entries and system responses show how you can output the contents of a file named **list1** using the default form of the **od** command. This form of the command outputs octal words. The **cat** command is first used to display the normal ASCII contents of the file.

```
$ cat list1<CR>
eggs
bread
milk
butter
meat
$ od list1<CR>
0000000 062547 063563 005142 071145 060544 005155 064554 065412
0000020 061165 072164 062562 005155 062541 072012
0000034
$
```

The following command line entries and system responses show how you can output the contents of a file named **list1** using the **-b** option of the command. This form of the command outputs the octal value for each character (byte). An *offset* of 27 bytes is used in this example. This *offset* causes the output to start at the last line of the file in this example. The **cat** command is first used to display the normal ASCII contents of the file. You need to refer to an octal map of the ASCII character set to make sense out of the output. For example, the letter "m" is octal 155; the letter "e" is octal 145; a new-line character is octal 012.

```
$ cat list1<CR>
eggs
bread
milk
butter
meat
$ od -b list1 27<CR>
0000027 155 145 141 164 012 000
0000034
$
```

"pack" — Compress Files

General

The **pack** command is used to compress and store files. Text files can be reduced between 60% and 75% of their original size. Load modules which use a larger character set and have a more uniform distribution of characters can be reduced to about 90% of their original size. The original file is removed and the compressed data is stored in a file with the same file name, except that a **.z** is added to the end of the file name. For example, if you compressed a file named **file1**, the compressed data will be stored in **file1.z**. The access modes, access and modified dates, and owner will remain the same as the original file. The compressed file can be restored to its original form using the **pcat** or **unpack** command.

How much a file is compressed depends on two things. They are:

1. The size of the input file
2. The character frequency distribution.

Usually, it is not worthwhile to compress files smaller than three blocks of data because a decoding tree is placed at the beginning of the compressed file. However, if the character frequency distribution is very skewed, you may wish to compress the file even if the file is less than three blocks. Some reasons for the character frequency distribution being skewed are printer plots, pictures or tables in the file.

The **pack** command will not work if:

1. The file appears to be already compressed.
2. The file name has more than 12 characters.

3. The file is linked to another file.
4. The file is a directory.
5. The file cannot be opened.
6. No disk storage blocks will be saved by compression.
7. A file called *name*.**z** already exists.
8. The **.z** file cannot be created.
9. An input/output error occurred during processing.

The **pack** command returns a value that is the number of files that it failed to compress.

Command Format

The general format of the **pack** command is as follows.

pack [-] *name* ...

where:

-	Used to set an internal flag that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of - in place of *name* will cause the internal flag to be set and reset.
name	The name of the file to be compressed.

Sample Command Use

The following command line entries and system responses show the basic operation of the **pack** command. The **ls -l** command is used to show what are the file sizes. The **pack** command is used to show how the command is input and the response you will receive. The **ls -l** command is used again to show the size of the files after they are compressed.

```
$ ls -l<CR>
total 109
-rw-------   1 cec    other        29727 Apr 11 13:14 file1
-rw-------   1 cec    other        24355 Apr 11 13:15 file2
$ pack - file1 file2<CR>
pack: file1: 36.2% Compression
                 from 29727 to 18980 bytes
                 Huffman tree has 15 levels below root
                 90 distinct bytes in input
                 dictionary overhead = 112 bytes
                 effective  entropy  = 5.11 bits/byte
                 asymptotic entropy  = 5.08 bits/byte
pack: file2: 36.2% Compression
                 from 24355 to 15530 bytes
                 Huffman tree has 15 levels below root
                 85 distinct bytes in input
                 dictionary overhead = 107 bytes
                 effective  entropy  = 5.10 bits/byte
                 asymptotic entropy  = 5.07 bits/byte
$ ls -l<CR>
total 73
-rw-------   1 cec    other        18980 Apr 11 13:14 file1.z
-rw-------   1 cec    other        15530 Apr 11 13:15 file2.z
$
```

"paste" — Side-by-Side File Merge

General

The **paste** command is used to combine two or more files of data in a side-by-side fashion. Each file of data is treated like a column of data in a table. The output of the paste command can be displayed on the terminal, redirected to a file, or redirected to another command.

Command Formats

Three general forms of the **paste** command are provided. These forms are as follows.

paste *file(s)*

paste -d *'list' file(s)*

paste -s [**-d** *'list'*] *file(s)*

The *file(s)* argument identifies the names of the files that are to be pasted together. The hyphen (-) can be used as a file name to read a line from the standard input. There is no prompting associated with the use of the hyphen.

The **-s** option is used to merge subsequent lines from each input file as opposed to one line from each input file.

The **-d***list* argument option is used to define the delimiter(s) that is used between the merged lines. The tab character is the default delimiter. The *list* argument identifies what is to replace the tab delimiter. The items (characters) identified in the *list* argument are used in sequence until the end of the list. Then, the listed delimiters are reused in the same sequence. In general, the **list** argument should be in double quotes.

For example, to get one backslash, use **-d'\'** as the argument; use **-d' '** as the argument to define a space as the delimiter. Special characters are defined by escape sequences. These include the following:

- **n** for new-line character
- **t** for tab character
- \\\\ for the backslash character
- **0** for an empty string.

Sample Command Use

The following examples are based on the use of two files named **list1** and **list2**. The contents of these two files are as follows.

```
list1:          list2:

list1: item1    list2: item1
list1: item2    list2: item2
list1: item3    list2: item3
list1: item4    list2: item4
list1: item5    list2: item5
list1: item6    list2: item6
list1: item7    list2: item7
```

The following command line entries and system responses show how you can merge the contents of two files using the simplest form of the **paste** command. This form of the command requires no options. The named files are merged in a side-by-side fashion with a tab character as the delimiter between the lines of the files. The output of the **paste** command is redirected to a file named **save**. The **cat** command is used to display the contents of the resulting file.

```
$ paste list1 list2 > save<CR>
$ cat save<CR>
list1: item1     list2: item1
list1: item2     list2: item2
list1: item3     list2: item3
list1: item4     list2: item4
list1: item5     list2: item5
list1: item6     list2: item6
list1: item7     list2: item7
$
```

The following command line entries and system responses show how you can merge the contents of two files using the form of the **paste -d***list* command. The named files are merged in a side-by-side fashion with a slash (/) character as the delimiter between the lines of the file. The output of the **paste** command is redirected to a file named **save**. The **cat** command is used to display the contents of the resulting file.

```
$ paste -d'/' list1 list2 > save<CR>
$ cat save<CR>
list1: item1/list2: item1
list1: item2/list2: item2
list1: item3/list2: item3
list1: item4/list2: item4
list1: item5/list2: item5
list1: item6/list2: item6
list1: item7/list2: item7
$
```

"pcat" — Concatenate and Print Packed Files

General

The **pcat** command is used to concatenate and print files that have been compressed by the **pack** command. The compressed file is expanded and printed on your terminal screen in its original form.

The **pcat** command will not work if:

1. The file name (exclusive of **.z**) has more than 12 characters.
2. The file cannot be opened.
3. The file does not appear to be the output of the **pack** command.

The **pcat** command returns a value that is the number of files that it failed to expand.

Command Format

The general format of the **pcat** command is as follows.

pcat *name ...*

The *name* argument identifies the name of the file that needs to be expanded. The **.z** at the end of the file name does not need to be input when specifying *name*.

The standard output of the **pcat** command can be redirected to a file. You will have two files: one that contains the compressed data (*name***.z**) and one that contains the original data. The general format when redirecting the output of the **pcat** command follows.

pcat *name* > *new.file*

where:

name	Identifies the name of the file that needs to be expanded.
new.file	Identifies the name of the file that contains the expanded data.

Sample Command Use

The following command line entries and system responses show the basic operation of the **pcat** command. The **ls -l** command is used to display the compressed files before the **pcat** command is given. The **pcat** command is used to show you how the command is input using the redirection method. The **ls -l** command is used again to display the results of executing the **pcat** command.

```
$ ls -l<CR>
total 71
-rw-------    1 cec    other        18980 Apr 11 13:14 file1.z
-rw-------    1 cec    other        15530 Apr 11 13:15 file2.z
$ pcat file1 > new.file1<CR>
$ pcat file2 > new.file2<CR>
$ ls -l<CR>
total 181
-rw-------    1 cec    other        18980 Apr 11 13:14 file1.z
-rw-------    1 cec    other        15530 Apr 11 13:15 file2.z
-rw-------    1 cec    other        29727 Apr 12 07:47 new.file1
-rw-------    1 cec    other        24355 Apr 12 07:48 new.file2
$
```

"pg" — Command Description

General

The **pg** command is a filter that will allow you to view a file a page at a time on a soft-copy terminal screen. A prompt (**:**) is displayed after every page. If a carriage return is entered after the prompt, another page is displayed. Other options, listed in this section, may be chosen. What makes the **pg** command different from other similar commands is that the **pg** command allows you to back up and review something that has already passed. The **pg** command scans the **terminfo** data base for your terminal type to determine the terminal attributes. The variable **TERM** specifies your terminal type. If **TERM** is not specified, the terminal type **dumb** is assumed. Refer to the *3B2 Computer Programmer Reference Manual* for information on the **terminfo** data base.

A pause will occur after each page is displayed and the prompt is given. There are three categories of responses that can be given when the prompt is displayed. The three categories are those that cause further perusal, those that search, and those that modify the perusal environment.

Commands which cause further perusal normally take a preceding *address*. The *address* is an optionally signed number which indicates the point from which further text should be displayed. The *address* can be given in pages or lines. A signed *address* specifies a point relative to the current page or line. An unsigned *address* specifies an address relative to the beginning of the file.

The perusal commands are as follows.

<newline> or *<blank>*
Display the next page. If a signed *address* is used, the **pg** command goes forward (+) or backward (-) the numbered amount of pages specified and displays that page on your terminal screen. If an unsigned *address* is used, the page number specified will be displayed.

l Scroll one line forward. If a signed *address* is used, the **pg** command simulates scrolling the screen, forward (+) or backward (-), the number of lines specified. If an unsigned *address* is used, the **pg** command prints a screenful beginning at the line number specified.

d or **^D** Simulates scrolling half a screen forward (+**1** *address*) or half a screen backward (-**1** *address*).

The next two perusal commands do not use addresses.

. or **^L** Causes the current page to be redisplayed.

$ Displays the last window (page) in the file. If the input is a pipe, use with caution.

The following commands are available for searching for specific patterns of text. These commands must be terminated by a *<newline>*, even if the *-n* option is specified. You may use the regular expressions of the **ed** command. Refer to the *UNIX System V User Guide* for information about the **ed** command.

i/pattern/	Search forward for the *i*th (default is *i*=1) occurrence of *pattern*. Searching will begin after the current page and will continue until the end of the file is reached. If the entire *pattern* is not on the same line, *pattern* will not be found.
i?pattern? or *î pattern^*	Search backward for the *i*th (default is *i*=1) occurrence of *pattern*. Searching will begin before the current page and will continue until the beginning of the file is reached. Use *î pattern^* if using an Adds 100 terminal.

The line found at the top of the screen will be displayed after the search has ended. By appending **m** or **b** to the search command, you can display the line at the middle of the window or the bottom of the window. The suffix **t** can be used to restore the original situation.

You can modify the perusal environment with the following commands.

*i***n**	Begin perusing the *i*th next file in the command line. If *i* is not specified, 1 is used.
*i***p**	Begin perusing the *i*th previous file in the command line. If *i* is not specified, 1 is used.
*i***w**	Display another window of text. If *i* is present, set the window size to *i*.
s *filename*	Save the current file that is being perused in *filename*. This command must be terminated

	by a <*newline*>, even if the *-n* option is specified.
h	Help command. An abbreviated summary of available commands is displayed.
q or **Q**	Quit the **pg** command.
!*command*	The *command* is executed by the shell. If the **SHELL** environment variable is set, that shell is used. If the **SHELL** environment variable is not set, the default shell is used. This command must be terminated by a <*newline*>, even if the *-n* option is specified.

You can stop sending output to the terminal at any time by depressing the quit key (normally control-\) or the interrupt (break) key. The prompt will appear and you may then enter commands in the normal manner. Unfortunately, some output is lost when you stop the output. This happens because any characters waiting in the terminal output queue are flushed when the quit signal occurs.

The **pg** command acts like the **cat** command if the standard output is not a terminal screen. The only difference is that a header is printed before each file if there is more than one file. Refer to the *UNIX System V User Guide* for information about the **cat** command.

Execution of the **pg** command is terminated if **BREAK**, **DEL**, or is depressed while the **pg** command is waiting for terminal input. If you are between prompts, these signals interrupt the current task and will place you in the prompt mode. Use the interrupt signals with caution when the input is coming from a pipe, since the interrupt is likely to terminate the other commands in the pipeline.

There are a couple of bugs that you need to know about. The first one is that the terminal tabs should be set to every eight positions or you may get undesirable results. The second one is that when using the **pg** command as a filter with another command that changes the terminal input/output options, terminal settings may not be restored correctly.

Command Format

The general format of the **pg** command is as follows.

pg [*-number*] [**-p** *string*] [**-cefns**] [*+linenumber*] [*+/pattern/*] [*file(s)*]

where:

-number	The size (number of lines) of the window. If the size is not specified, the default value is one line less than the total number of lines that can be displayed on your terminal screen.
-p *string*	Causes *string* to be used as the prompt. If **%d** appears in *string*, the first occurrence of **%d** in the prompt is replaced by the current page number when the prompt is issued.
-c	Take cursor to the home position and clear the screen before displaying a page. If **clear_screen** is not defined in the **terminfo** data base, this option will be ignored.
-e	Normally, a pause will occur at the end of each page and at the end of each file. This option eliminates the pause at the end of each file.
-f	Normally, if a line is longer than the terminal screen width, it is split into two lines. However, there are times when some sequences of characters in the text generate undesirable results; such as escape sequences for underlining. In this case, you can use this option to inhibit the **pg**

	command from splitting lines.
-n	Normally, commands must be terminated by a *<newline>* character. This option causes the command to end as soon as a command letter is entered.
-s	Causes all messages and prompts to be printed in the standout mode (usually inverse video).
+*linenumber*	Start up at *linenumber*.
+/*pattern*/	Start up at the first line containing the pattern specified.
file(s)	The name of the file to be examined. If *file(s)* is not specified or if a minus sign (**-**) is specified, the **pg** command reads the standard input.

Sample Command Use

The following command line entry and system response show the basic operation of the **pg** command. The **pg** command along with the **news** command is used in a pipeline to read the system news.

```
$ news | pg -p "(Page %d):"<CR>
Note: The first page of the news will appear next.
(Page 1): Note: This is the prompt. It will appear
          after each page with the number of the page
          you are on. You may now enter one of the
          commands that manipulate the text or enter
          q to quit.
$
```

"sdiff" — Side-By-Side Difference Program

General

The **sdiff** command uses the output of the **diff** command (discussed earlier in this chapter) to produce a side-by-side listing of two files. If the lines are identical, each line of the two files are printed side-by-side with a blank gutter between the two files. If the line exists only in *file1*, a less than (<) symbol is in the gutter. If the line exists only in *file2*, a greater than (>) is in the gutter. If the line exists in both files and they are different, a pipe symbol (¦) is in the gutter.

For example:

```
x   ¦   y
a       a
b   <
c   <
d       d
    >   c
```

Command Format

The general format of the **sdiff** command is as follows.

sdiff [*options ...*] *file1 file2*

The following options exist:

-w *n*	Use the next argument (*n*) as the width of the output line. If *n* is not specified, the line length will be 130 characters.
-l	Only print the left side of any lines that are identical.

-s	Do not print identical lines.
-o *output*	Use the next argument (*output*) to create a third file that will let you control the merging of *file1* and *file2*. All identical lines of *file1* and *file2* are copied to the *output* file. All different lines of *file1* and *file2* are printed on your terminal screen. After the different lines are printed, you will receive a prompt (**%**). After the prompt (**%**) is received, enter one of the following commands.

l	Append the left column to the output file.
r	Append the right column to the output file.
s	Turn the silent mode on; do not print identical lines.
v	Turn the silent mode off.
e l	Will let you edit the left column.
e r	Will let you edit the right column.
e b	Will let you edit the concatenation of the left and right columns.
e	Will let you edit a new file.
q	Exit from the program.

When you exit from the editor, the resulting file is concatenated on the end of the *output* file.

The arguments *file1* and *file2* are the files that are being compared.

Sample Command Use

The following command line entries and system responses show the basic operation of the **sdiff** command. The **cat** command is used to display the contents of file1 and file2. The **sdiff** command is then used to display a side-by-side comparison of file1 and file2.

```
$ cat file1<CR>
1000
2000
4000
8000
16000
32000
64000
128000
$ cat file2<CR>
500
1000
2000
3000
8000
16000
34000
64000
$ sdiff -w 30 file1 file2<CR>
              >  500
1000              1000
2000              2000
4000          |  3000
8000              8000
16000             16000
32000         |  34000
64000             64000
128000        <
$
```

"split" — Split a File Into Pieces

General

The **split** command is used to read a file and write it in *n* number of lines onto a set of output files. Default is 1000 lines per file. The name of the first output file is *name* with **aa** through **zz** appended. The output file will be appended with **aa**, then **ab**, then **ac**, and so forth until **zz** is reached. A maximum of 676 files can be created using the **split** command. There must not be more than 12 characters in *name*. If no output name is given, **x** is default.

Command Format

The general format of the **split** command is as follows.

split [*-n*] [*file* [*name*]]

where:

-n	The number of lines that are to be written onto each output file.
file	The name of the file to be split.
name	The name of the output file.

If no input file is given or if **-** is given, the standard input is used.

Sample Command Use

The following command line entries and system responses show the basic operation of the **split** command. The **cat** command is used to display the contents of **file1**. The **split** command is used to split **file1** into 3 lines per output file. The **ls -l** command is used to display the new files that are created. The **cat** command is used again to display the contents of each one of the new files.

```
$ cat file1<CR>
This will be the first line of the first output file.
...
...
This will be the first line of the second output file.
...
...
This will be the first line of the third output file.
...
...
$ split -3 file1 new.file1<CR>
$ ls -l<CR>
total 5
-rw-------   1 cec   other        187 Apr 17 10:52 file1
-rw-------   1 cec   other         62 Apr 17 10:53 new.file1aa
-rw-------   1 cec   other         63 Apr 17 10:53 new.file1ab
-rw-------   1 cec   other         62 Apr 17 10:53 new.file1ac
$ cat new.file1aa<CR>
This will be the first line of the first output file.
...
...
$ cat new.file1ab<CR>
This will be the first line of the second output file.
...
...
$ cat new.file1ac<CR>
This will be the first line of the third output file.
...
...
$
```

"sum" — Print Check Sum and Block Count of a File

General

The **sum** command is used to calculate and output a 16-bit checksum for a specified file. Typically, the command is used to look for bad data or to validate a file transmitted over a communications interface. The number of blocks in the specified file is also output.

To use the **sum** command to validate transmitted data, the checksum is executed on the file before transmission and the results are sent to the destination. At the destination, the **sum** command is again executed on the received data. The before and after checksums are then compared. Matching checksums indicate a successful transfer of data; a mismatch indicates a problem. Note that you must know whether or not to use the **-r** option when validating transferred data. You must use the same form of the command to calculate the checksum at the source and destination in order to be able to validate the transmitted data.

Command Format

The general format of the **sum** command is as follows.

sum [**-r**] *file*

The **-r** option causes the command to use a different rationale (algorithm) in computing the checksum. The *file* argument identifies the name of the file to be processed. Note that the file name can be expressed as a complete path name.

Sample Command Use

The following command line entries and system responses show you a typical **sum** command output. The first field output is the checksum, followed by the number of blocks (1), followed by the name of the file (**list1**).

```
$ sum list1<CR>
2496 1 list1
$ sum -r list1<CR>
55792     1 list1
$
```

"tail" — Output End of a File

General

The **tail** command is used to output the last portion of some data. The source data operated on by the command can be from a file, the output of another command, or from the terminal. Options are provided to tell the command at what point from the beginning or end of the input data to start passing data to the output. The start can be expressed in the number of lines, blocks, or characters from the beginning or end of the data.

Command Format

The general format of the **tail** command is as follows.

tail [±[*number*][**lbc**[**f**]]] [*file*]

The *number* argument identifies the number of units from the beginning or from the end of the input where the output is to begin. A plus sign preceding the number means from the beginning of the input data. A minus sign preceding the number means from the end of the input. The units used for the *number* argument are lines (**l**), blocks (**b**), or characters (**c**). The unit identifier immediately follows the *number* argument (no space).

The **-f** option is used to continuously read data from a file. The option provides the ability to monitor the growth of a file that is being written by some other process. The **-f** option is not applicable when data is being piped to the **tail** command.

The *file* argument identifies the name of the the source file. Note that the file name can be expressed as a complete path name.

Sample Command Use

The following examples are based on the contents of a file named **sample**. The contents of this file are as follows.

line 1
line 2
line 3
line 4
line 5
line 6
line 7
line 8
line 9
line 10
line 11
line 12

The following command line entries and system responses show how you can output the end of a file of data using the simplist form of the **tail** command. This form of the command outputs the last ten lines of the contents of the **sample** file. The default of the *number* argument is **-10l**.

```
$ tail sample<CR>
line 3
line 4
line 5
line 6
line 7
line 8
line 9
line 10
line 11
line 12
$
```

The following sample command lines and system responses show you how to output the contents of the **sample** file starting 45 characters from the beginning of the file.

```
$ tail +45c sample<CR>
ne 7
line 8
line 9
line 10
line 11
line 12
$
```

"tr" — Translate Characters

General

The **tr** command is used as a filter to modify data that is passed through it on a character basis. The command functions like a stream editor. Repeated occurrences of a character in succession can be reduced to a single occurrence of the character. Characters can be identified by ASCII letter or octal value. Octal values are preceded by a backslash (\). Ranges of characters are identified by enclosing the range in brackets. For example, [**a-c**] represents the letters a, b, and c. The entire lowercase ASCII range is identified by [**a-z**]. Multiple occurrences of a character is represented by an expression [**x*n**], where the **x** is any character and the **n** is the number of repetitions of **x**. The number is treated as an octal number if the most significant digit is a zero. The number is treated as a decimal number if the most significant digit is other than a zero.

Command Format

The general format of the **tr** command is as follows.

tr [**-cds**] [*string1* [*string2*]]

The **-c** option reverses the meaning of *string1* in relation to translating characters to *string2*. The *string1* argument identifies the characters that ARE NOT to be translated. The characters identified in the *string1* argument pass unchanged to the output. When the **-c** option is omitted, *string1* characters are translated to *string2* characters.

The **-d** option causes the characters identified by *string1* to be deleted from the output. The *string2* argument is not used with the **-d** option. If the *string2* argument is provided, it will be ignored.

The **-s** option causes multiple occurrences of the characters identified by *string2* to be replaced by a single occurrence of the characters. If only one string argument is given, then *string1* defines the character(s) to be operated on by the command.

The *string1* argument identifies input characters that are to be operated on by the command. When a *string2* argument is also provided, the characters found in *string1* are mapped to the corresponding character in *string2*.

Sample Command Use

The following command line entries and system responses show how you can change all lowercase letters in a file named **list1** to uppercase letters. The **cat** command is used to display the contents of **list1**. The output of the **tr** command is redirected to a file named **LIST**.

```
$ cat list1<CR>
eggs
bread
milk
butter
meat
$ tr "[a-z]" "[A-Z]" < list1 > LIST<CR>
$ cat LIST<CR>
EGGS
BREAD
MILK
BUTTER
MEAT
$
```

The following command line entries and system responses show how you can use the **tr** command to reduce multiple consecutive occurrences of a space character to a single occurrence throughout a file of data. The **cat** command is used to display the contents of the files.

```
$ cat list3<CR>
This    file  contains  lines
with multiple    spaces  between words.
$ tr -s " "  < list3 > newlist<CR>
$ cat newlist<CR>
This file contains lines
with multiple spaces between words.
$
```

The following command line entries and system responses show how you can use the **tr** command to put each word in a file on a separate line. The **cat** command is used to display the contents of the sample file. The output of the **tr** command is displayed on the terminal in this example.

```
$ cat file<CR>
This file contains one line of text.
$ tr -cs "[A-z]" "[\012*]" < file<CR>
This
file
contains
one
line
of
text
$
```

"uniq" — Report Repeated Lines in a File

General

The **uniq** command is used to read an input file while comparing adjacent lines. The second and succeeding copies of repeated lines are removed in the normal output mode and the remaining lines are written on the output file. Repeated lines must be adjacent in order to be found. The input and output files must have different names.

Command Format

The general format of the **uniq** command is as follows.

uniq [*-udc* [*+n*] [*-n*]] [*input* [*output*]]

where:

-u	The lines that are not repeated in the input file are written on the output file.
-d	Only one copy of just the repeated lines is written on the output file.
-c	The output file is generated in the normal output mode with a count of the number of times each line occurs. This option supersedes *-u* and *-d*.
+n	*n* amount of characters are ignored. Fields are skipped before characters. A field is defined as a string of nonspace, nontab characters separated by tabs and spaces from its neighbors.
-n	The first *n* amount of fields together with any blanks before each field are ignored.

input	The name of the input file.
output	The name of the output file.

The normal output mode is the union of the *-u* and *-d* options.

Sample Command Use

The following command line entries and system responses show the basic operation of the **uniq** command. The **cat** command is used to display a grocery list. The **sort** command is used to alphabetize the grocery list. To learn more about the **sort** command, refer to your *UNIX System V User Guide*. The **cat** command is used again to display the alphabetized grocery list. The **uniq** command is used to remove the repeated lines and to count the number of times each item was listed. The **cat** command is used again to display the results.

```
$ cat file1<CR>
bread
milk
butter
ice cream
meat
milk
vegetables
potato chips
drinks
orange juice
bread
vegetables
$ sort file1 > file2<CR>
$ cat file2<CR>
bread
bread
butter
drinks
ice cream
meat
milk
milk
orange juice
potato chips
vegetables
vegetables
$ uniq -c file2 file3<CR>
$ cat file3<CR>
   2 bread
   1 butter
   1 drinks
   1 ice cream
   1 meat
   2 milk
   1 orange juice
   1 potato chips
   2 vegetables
$
```

"unpack" — Expand Files

General

The **unpack** command is used to expand files created by the **pack** command. The compressed data is expanded to its original form. The compressed file is removed and the expanded data is placed in a file with the same file name, except that the **.z** is dropped. For example, if you expanded a file named **file1.z**, the expanded data will be placed in **file1**. The access modes, access and modified dates, and owner will remain the same as the compressed file.

The **unpack** command will not work if:

1. The file name (exclusive of **.z**) has more than 12 characters.
2. The file cannot be opened.
3. The file does not appear to be the output of the **pack** command.
4. A file with the " unpacked" name already exists.
5. The unpacked file cannot be created.

The **unpack** command returns a value that is the number of files that it failed to expand.

Command Format

The general format of the **unpack** command is as follows.

unpack *name ...*

The *name* argument identifies the name of the file that needs to be expanded. The **.z** at the end of the file name does not need to be input when specifying *name*.

Sample Command Use

The following command line entries and system responses show the basic operation of the **unpack** command. The **ls -l** command is used to display the character size of the compressed files before they are expanded. The **unpack** command is used to expand the compressed files. The **ls -l** command is used again to display the character size of the expanded files.

```
$ ls -l<CR>
total 71
-rw-------   1 cec   other       18980 Apr 11 13:14 file1.z
-rw-------   1 cec   other       15530 Apr 11 13:15 file2.z
$ unpack file1 file2<CR>
unpack: file1: unpacked
unpack: file2: unpacked
$ ls -l<CR>
total 110
-rw-------   1 cec   other       29727 Apr 11 13:14 file1
-rw-------   1 cec   other       24355 Apr 11 13:15 file2
$
```

Appendix

MANUAL PAGES

This appendix contains the UNIX System manual pages for Directory and File Management Utilities. Manual pages for the following commands are provided in alphabetical sequence.

awk	dircmp	nl	split
bdiff	egrep	od	sum
bfs	fgrep	pack	tail
comm	file	paste	tr
csplit	find	pcat	uniq
cut	join	pg	unpack
diff3	newform	sdiff	

Certain of these commands are described on a manual page along with one or more other commands. Commands that are described on a different manual page than indicated by the name of the command are as follows.

COMMAND	MANUAL PAGE
pcat	pack
unpack	pack

For your convenience, the user manual pages for the Directory and File Management Utilities are provided in both this guide and alphabetically in the *3B2 Computer User Reference Manual.*

NAME

ar − archive and library maintainer for portable archives

SYNOPSIS

ar key [posname] afile [name] ...

DESCRIPTION

The *Ar* command maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. It can be used, though, for any similar purpose. The magic string and the file headers used by *ar* consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

When *ar* creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure is described in detail in *ar*(4). The archive symbol table [described in *ar*(4)] is used by the link editor [*ld*(1)] to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is only created and maintained by *ar* when there is at least one object file in the archive. The archive symbol table is in a specially named file which is always the first file in the archive. This file is never mentioned or accessible to the user. Whenever the *ar*(1) command is used to create or update the contents of such an archive, the symbol table is rebuilt. The **s** option described below will force the symbol table to be rebuilt.

Key is an optional −, followed by one character from the set **drqtpmx**, optionally concatenated with one or more of **vuaibcls**. *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If the optional character **u** is used with **r**, then only those files with dates of modification later than the archive files are replaced. If an optional positioning character from the set **abi** is used, then the *posname* argument must be present and specifies that new files are to be placed after (**a**) or before (**b** or **i**) *posname*. Otherwise new files are placed at the end.

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece-by-piece.

t Print a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Print the named files in the archive.

m Move the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in **r**, specifies where the files are to be moved.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v Give a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, give a long listing of all information about the files. When used with **x**, precede each file with a name.

c Suppress the message that is produced by default when *afile* is created.

l	Place temporary files in the local current working directory, rather than in the directory specified by the environment variable **TMPDIR** or in the default directory **/tmp**.
s	Force the regeneration of the archive symbol table even if *ar*(1) is not invoked with a command which will modify the archive contents. This command is useful to restore the archive symbol table after the *strip*(1) command has been used on the archive.

FILES

/tmp/ar*	temporaries

SEE ALSO

ld(1), lorder(1), strip(1).
tmpnam(3S), a.out(4), ar(4) in the *3B2 Computer System Programmer Reference Manual.*

BUGS

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

awk – pattern scanning and processing language

SYNOPSIS

awk [**–Fc**] [prog] [parameters] [files]

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as **–f** *file*. The *prog* string should be enclosed in single quotes (′) to protect it from the shell.

Parameters, in the form x=... y=... etc., may be passed to *awk*.

Files are read in order; if there are no files, the standard input is read. The file name – means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS; see below). The fields are denoted **$1**, **$2**, ...; **$0** refers to the entire line.

A pattern-action statement has the form:

```
pattern { action }
```

A missing action means print the line; a missing pattern always matches. An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next     # skip remaining patterns on this input line
exit     # skip the rest of the input
```

Statements are terminated by semicolons, new-lines, or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, –, *, /, %, and concatenation (indicated by a blank). The **C** operators ++, – –, +=, – =, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted (").

The *print* statement prints its arguments on the standard output (or on a file if >*expr* is present), separated by the current output field separator, and terminated by the output record separator. The *printf* statement formats its expression list according to the format [see *printf*(3S)*inthe3B2*Computer

The built-in function *length* returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions *exp*, *log*, *sqrt*, and *int*. The last truncates its argument to an integer; *substr*(*s*, *m*, *n*) returns the *n*-character substring of *s* that begins at position *m*. The function *sprintf*(*fmt*, *expr*, *expr*, ...) formats the expressions according to the *printf*(3S) format given by *fmt* and returns the resulting string.

Patterns are arbitrary Boolean combinations (**!**, **| |**, **& &**, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in *egrep* (see *grep*(1)). Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions. A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
```

where a relop is any of the six relational operators in C, and a matchop is either ˜ (for *contains*) or !˜ (for *does not contain*). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character *c* may be used to separate the fields by starting the program with:

```
BEGIN { FS = c }
```

or by using the **−F***c* option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default new-line); and OFMT, the output format for numbers (default **%.6g**).

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

```
        { s += $1 }
END     { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

```
{ for (i = NF; i > 0; −−i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

Print file, filling in page numbers starting at 5:

```
/Page/  { $2 = n++; }
        { print }
```

command line: awk −f program n=5 input

SEE ALSO

grep(1), sed(1).
malloc(3X), printf(3S) in the *3B2 Computer System Programmer Reference Manual.*
lex(1) in the *3B2 Computer System Extended Software Generation System Utilities.*

BUGS

Input white space is not preserved on output if fields are involved.
There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string ("") to it.

NAME

bdiff − big diff

SYNOPSIS

bdiff file1 file2 [n] [−s]

DESCRIPTION

Bdiff is used in a manner analogous to *diff*(1) to find which lines must be changed in two files to bring them into agreement. Its purpose is to allow processing of files which are too large for *diff*.

The parameters to *bdiff* are:

file1 (file2)
: The name of a file to be used. If *file1 (file2)* is −, the standard input is read.

n
: The number of line segments. The value of *n* is 3500 by default. If the optional third argument is given and it is numeric, it is used as the value for *n*. This is useful in those cases in which 3500-line segments are too large for *diff*, causing it to fail.

−s
: Specifies that no diagnostics are to be printed by *bdiff* (silent option). Note, however, that this does not suppress possible exclamations by *diff*.

Bdiff ignores lines common to the beginning of both files, splits the remainder of each file into *n*-line segments, and invokes *diff* upon corresponding segments. If both optional arguments are specified, they must appear in the order indicated above.

The output of *bdiff* is exactly that of *diff*, with line numbers adjusted to account for the segmenting of the files (that is, to make it look as if the files had been processed whole). Note that because of the segmenting of the files, *bdiff* does not necessarily find a smallest sufficient set of file differences.

FILES

/tmp/bd?????

SEE ALSO

diff(1), help(1).

DIAGNOSTICS

Use *help*(1) for explanations.

NAME

bfs − big file scanner

SYNOPSIS

bfs [−] name

DESCRIPTION

The *Bfs* command is (almost) like *ed*(1) except that it is read-only and processes much larger files. Files can be up to 1024K bytes (the maximum possible size) and 32K lines, with up to 512 characters, including new-line, per line (255 for 16-bit machines). *Bfs* is usually more efficient than *ed* for scanning a file, since the file is not copied to a buffer. It is most useful for identifying sections of a large file where *csplit*(1) can be used to divide it into more manageable pieces for editing.

Normally, the size of the file being scanned is printed, as is the size of any file written with the **w** command. The optional − suppresses printing of sizes. Input is prompted with * if **P** and a carriage return are typed as in *ed*. Prompting can be turned off again by inputting another **P** and carriage return. Note that messages are given in response to errors if prompting is turned on.

All address expressions described under *ed* are supported. In addition, regular expressions may be surrounded with two symbols besides / and ?: > indicates downward search without wrap-around, and < indicates upward search without wrap-around. There is a slight difference in mark names: only the letters **a** through **z** may be used, and all 26 marks are remembered.

The **e**, **g**, **v**, **k**, **p**, **q**, **w**, **=**, **!** and null commands operate as described under *ed*. Commands such as **−−−**, **+++−**, **+++=**, **−12**, and **+4p** are accepted. Note that **1,10p** and **1,10** will both print the first ten lines. The **f** command only prints the name of the file being scanned; there is no *remembered* file name. The **w** command is independent of output diversion, truncation, or crunching (see the **xo**, **xt** and **xc** commands, below). The following additional commands are available:

xf *file*
: Further commands are taken from the named *file*. When an end-of-file is reached, an interrupt signal is received or an error occurs, reading resumes with the file containing the **xf**. The **xf** commands may be nested to a depth of 10.

xn
: List the marks currently in use (marks are set by the **k** command).

xo [*file*]
: Further output from the **p** and null commands is diverted to the named *file*, which, if necessary, is created mode 666. If *file* is missing, output is diverted to the standard output. Note that each diversion causes truncation or creation of the file.

: *label*
: This positions a *label* in a command file. The *label* is terminated by new-line, and blanks between the : and the start of the *label* are ignored. This command may also be used to insert comments into a command file, since labels need not be referenced.

(. , .)**xb**/*regular expression*/*label*

A jump (either upward or downward) is made to *label* if the command succeeds. It fails under any of the following conditions:

1. Either address is not between **1** and **$**.
2. The second address is less than the first.
3. The regular expression does not match at least one line in the specified range, including the first and last lines.

On success, **.** is set to the line matched and a jump is made to *label*. This command is the only one that does not issue an error message on bad addresses, so it may be used to test whether addresses are bad before other commands are executed. Note that the command

```
xb/^/ label
```

is an unconditional jump.
The **xb** command is allowed only if it is read from someplace other than a terminal. If it is read from a pipe only a downward jump is possible.

xt *number*

Output from the **p** and null commands is truncated to at most *number* characters. The initial number is 255.

xv[*digit*][*spaces*][*value*]

The variable name is the specified *digit* following the **xv**. The commands **xv5100** or **xv5 100** both assign the value **100** to the variable **5**. The command **Xv61,100p** assigns the value **1,100p** to the variable **6**. To reference a variable, put a % in front of the variable name. For example, using the above assignments for variables **5** and **6**:

```
1,%5p
1,%5
%6
```

will all print the first 100 lines.

```
g/%5/p
```

would globally search for the characters **100** and print each line containing a match. To escape the special meaning of %, a \ must precede it.

```
g/".*\%[cds]/p
```

could be used to match and list lines containing *printf* of characters, decimal integers, or strings.

Another feature of the **xv** command is that the first line of output from a UNIX system command can be stored into a variable. The only requirement is that the first character of *value* be an **!**. For example:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
xv6!expr %6 + 1
```

would put the current line into variable **5**, print it, and increment the variable **6** by one. To escape the special meaning of **!** as the first character of *value*, precede it with a \.

```
xv7\!date
```

stores the value **!date** into variable **7**.

xbz *label*

xbn *label*

These two commands will test the last saved *return code* from the execution of a UNIX system command (!*command*) or nonzero value, respectively, to the specified label. The two examples below both search for the next five lines containing the string **size**.

```
xv55
: 1
/size/
xv5!expr %5 − 1
!if 0%5 != 0 exit 2
xbn 1
xv45
: 1
/size/
xv4!expr %4 − 1
!if 0%4 = 0 exit 2
xbz 1
```

xc [*switch*]

If *switch* is **1**, output from the **p** and null commands is crunched; if *switch* is **0** it is not. Without an argument, **xc** reverses *switch*. Initially *switch* is set for no crunching. Crunched output has strings of tabs and blanks reduced to one blank and blank lines suppressed.

SEE ALSO

csplit(1), ed(1).

DIAGNOSTICS

? for errors in commands, if prompting is turned off. Self-explanatory error messages when prompting is on.

NAME

comm − select or reject lines common to two sorted files

SYNOPSIS

comm [− [**123**]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence (see *sort*(1)), and produces a three-column output: lines only in *file1*; lines only in *file2*; and lines in both files. The file name − means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus **comm −12** prints only the lines common to the two files; **comm −23** prints only lines in the first file but not in the second; **comm −123** is a prints nothing.

SEE ALSO

cmp(1), diff(1), sort(1), uniq(1).

NAME

csplit − context split

SYNOPSIS

csplit [**−s**] [**−k**] [**−f** prefix] file arg1 [... argn]

DESCRIPTION

Csplit reads *file* and separates it into n+1 sections, defined by the arguments *arg1... argn*. By default the sections are placed in xx00 ... xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00:	From the start of *file* up to (but not including) the line referenced by *arg1*.
01:	From the line referenced by *arg1* up to the line referenced by *arg2*.
	⋮
n+1:	From the line referenced by *argn* to the end of *file*.

If the *file* argument is a − then standard input is used.

The options to *csplit* are:

−s	*Csplit* normally prints the character counts for each file created. If the **−s** option is present, *csplit* suppresses the printing of all character counts.
−k	*Csplit* normally removes created files if an error occurs. If the **−k** option is present, *csplit* leaves previously created files intact.
−f *prefix*	If the **−f** option is used, the created files are named *prefix***00** ... *prefixn*. The default is **xx00** ... **xx***n*.

The arguments (*arg1* ... *argn*) to *csplit* can be a combination of the following:

/rexp/	A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or − some number of lines (e.g., **/Page/−5**).
%rexp%	This argument is the same as */rexp/*, except that no file is created for the section.
lnno	A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.
{*num*}	Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the shell in the appropriate quotes. Regular expressions may not contain embedded new-lines. *Csplit* does not affect the original file; it is the users responsibility to remove it.

EXAMPLES

```
csplit −f cobol file '/procedure division/' /par5./ /par16./
```

This example creates four files, **cobol00** ... **cobol03**. After editing the "split" files, they can be recombined as follows:

```
cat cobol0[0−3] > file
```

Note that this example overwrites the original file.

```
csplit −k file  100  {99}
```

This example would split the file at every 100 lines, up to 10,000 lines. The **−k** option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

```
csplit −k prog.c '%main(%' '/^}/+1' {20}
```

Assuming that **prog.c** follows the normal **C** coding convention of ending routines with a } at the beginning of the line, this example will create a file containing each separate **C** routine (up to 21) in **prog.c**.

SEE ALSO

ed(1), sh(1).
regexp(5) in the *3B2 Computer System Programmer Reference Manual.*
regexp(5).

DIAGNOSTICS

Self-explanatory except for:

arg − out of range

which means that the given argument did not reference a line between the current position and the end of the file.

NAME

cut − cut out selected fields of each line of a file

SYNOPSIS

cut −**c**list [file1 file2 ...]
cut −**f**list [−**d**char] [−**s**] [file1 file2 ...]

DESCRIPTION

Use *cut* to cut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (−**c** option) or the length can vary from line to line and be marked with a field delimiter character like *tab* (−**f** option). *Cut* can be used as a filter; if no files are given, the standard input is used.

The meanings of the options are:

list	A comma-separated list of integer field numbers (in increasing order), with optional − to indicate ranges [e.g., **1,4,7**; **1−3,8**; **−5,10** (short for **1−5,10**); or **3−** (short for third through last field)].
−**c***list*	The *list* following −**c** (no space) specifies character positions (e.g., **−c1−72** would pass the first 72 characters of each line).
−**f***list*	The *list* following −**f** is a list of fields assumed to be separated in the file by a delimiter character (see −**d**); e.g., **−f1,7** copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless −**s** is specified.
−**d***char*	The character following −**d** is the field delimiter (−**f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.
−**s**	Suppresses lines with no delimiter characters in case of −**f** option. Unless specified, lines with no delimiters will be passed through untouched.

Either the −**c** or −**f** option must be specified.

Use *grep*(1) to make horizontal “cuts” (by context) through a file, or *paste*(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use *cut* and *paste*.

EXAMPLES

cut −d: −f1,5 /etc/passwd	mapping of user IDs to names
name=‵who am i | cut −f1 −d" "‵	to set **name** to current login name.

DIAGNOSTICS

line too long	A line can have no more than 1023 characters or fields.
bad list for c/f option	Missing −**c** or −**f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.
no fields	The *list* is empty.

SEE ALSO

grep(1), paste(1).

NAME

diff3 − 3-way differential file comparison

SYNOPSIS

diff3 [**−ex3**] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	*file1* is different
====2	*file2* is different
====3	*file3* is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1* **a**	Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.
f : *n1* , *n2* **c**	Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n*2, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the **−e** option, *diff3* publishes a script for the editor *ed* that will incorporate into *file1* all changes between *file2* and *file3*, i.e., the changes that normally would be flagged ==== and ====3. Option **−x** (**−3**) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed − file1
```

FILES

/tmp/d3∗
/usr/lib/diff3prog

SEE ALSO

diff(1).

BUGS

Text lines that consist of a single **.** will defeat **−e**.
Files longer than 64K bytes will not work.

NAME

dircmp — directory comparison

SYNOPSIS

dircmp [**−d**] [**−s**] [**−w***n*] dir1 dir2

DESCRIPTION

Dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all the options. If no option is entered, a list is output indicating whether the file names common to both directories have the same contents.

−d Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in *diff*(1).

−s Suppress messages about identical files.

−w*n* Change the width of the output line to *n* characters. The default width is 72.

SEE ALSO

cmp(1), diff(1).

NAME

egrep − search a file for a pattern

SYNOPSIS

egrep [options] [expression] [files]

DESCRIPTION

The *egrep* command searches the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Egrep* patterns are full regular *expressions*; it uses a fast deterministic algorithm that sometimes needs exponential space. The following *options* are recognized:

−v All lines but those matching are printed.
Same as a simple *expression* argument, but useful when the *expression* begins with a −

−f *file*
The regular *expression* is taken from the *file*.

In all cases, the file name is output if there is more than one input file. Care should be taken when using the characters $, *, [, ^, |, (,), and \ in *expression*, because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes '...'.

Egrep accepts regular expressions as in *ed*(1), except for \(and \), with the addition of:

1. A regular expression followed by + matches one or more occurrences of the regular expression.
2. A regular expression followed by ? matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by | or by a new-line match strings that are matched by either.
4. A regular expression may be enclosed in parentheses () for grouping.

The order of precedence of operators is **[]**, then * ? +, then concatenation, then | and new-line.

SEE ALSO

ed(1), fgrep(1), grep(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Lines are limited to 256 characters; longer lines are truncated.
Egrep does not recognize ranges, such as **[a−z]**, in character classes.

NAME

fgrep − search a file for a pattern

SYNOPSIS

fgrep [options] [strings] [files]

DESCRIPTION

Commands of the *frep* family search the input *files* (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. *Fgrep* patterns are fixed *strings*; it is fast and compact. The following *options* are recognized:

−v All lines but those matching are printed.
−x (Exact) only lines matched in their entirety are printed
−c Only a count of matching lines is printed.
−l Only the names of files with matching lines are listed (once), separated by new-lines.
−n Each line is preceded by its relative line number in the file.
−b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.
−e *expression*
Same as a simple *expression* argument, but useful when the *expression* begins with a **−**.
−f *file*
The regular *strings* list is taken from the *file*.

In all cases, the file name is output if there is more than one input file. Care should be taken when using the characters $, *, [, ^, |, (,), and \ in *expression*, because they are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes '...'.

Fgrep searches for lines that contain one of the *strings* separated by new-lines.

The order of precedence of operators is **[ ]**, then * ? +, then concatenation, then | and new-line.

SEE ALSO

ed(1), sed(1), sh(1).

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

BUGS

Ideally there should be only one *fgrep*, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs. Lines are limited to 256 characters; longer lines are truncated.

NAME

file — determine file type

SYNOPSIS

file [**−c**] [**−f** ffile] [**−m** mfile] arg ...

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ASCII, *file* examines the first 512 bytes and tries to guess its language. If an argument is an executable **a.out**, *file* will print the version stamp, provided it is greater than 0.

−c The −c option causes *file* to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under **−c**.

−f If the **−f** option is given, the next argument is taken to be a file containing the names of the files to be examined.

−m The **−m** option instructs *file* to use an alternate magic file.

File uses the file **/etc/magic** to identify files that have some sort of *magic number*, that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of **/etc/magic** explains its format.

NAME

find − find files

SYNOPSIS

find path-name-list expression

DESCRIPTION

Find recursively descends the directory hierarchy for each path name in the *path-name-list* (i.e., one or more path names) seeking files that match a boolean *expression* written in the primaries given below. In the descriptions, the argument *n* is used as a decimal integer where +*n* means more than *n*, −*n* means less than *n* and *n* means exactly *n*. Valid expressions are:

−name *file*	True if *file* matches the current file name. Normal shell argument syntax may be used if escaped (watch out for **[**, **?** and *).
−perm *onum*	True if the file permission flags exactly match the octal number *onum* (see *chmod*(1)). If *onum* is prefixed by a minus sign, more flag bits [017777, see *stat*(2)*in*the*3B2*Computer become significant and the flags are compared.
−type *c*	True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **p**, or **f** for block special file, character special file, directory, fifo (a.k.a named pipe), or plain file respectively.
−links *n*	True if the file has *n* links.
−user *uname*	True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the **/etc/passwd** file, it is taken as a user ID.
−group *gname*	True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the **/etc/group** file, it is taken as a group ID.
−size *n*[**c**]	True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a **c**, the size is in characters.
−atime *n*	True if the file has been accessed in *n* days. The access time of directories in *path-name-list* is changed by *find* itself.
−mtime *n*	True if the file has been modified in *n* days.
−ctime *n*	True if the file has been changed in *n* days.
−exec *cmd*	True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument **{}** is replaced by the current path name.
−ok *cmd*	Like **−exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing **y**.
−print	Always true; causes the current path name to be printed.
−cpio *device*	Always true; write the current file on *device* in *cpio*(1) format (5120-byte records).
−newer *file*	True if the current file has been modified more recently than the argument *file*.

−depth Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This can be useful when *find* is used with *cpio*(1) to transfer files that are contained in directories without write permission.

(*expression*) True if the parenthesized expression is true (parentheses are special to the shell and must be escaped).

The primaries may be combined using the following operators (in order of decreasing precedence):

1) The negation of a primary (**!** is the unary *not* operator).

2) Concatenation of primaries (the *and* operation is implied by the juxtaposition of two primaries).

3) Alternation of primaries (**−o** is the *or* operator).

EXAMPLE

To remove all files named **a.out** or ***.o** that have not been accessed for a week:

```
find / \( −name a.out −o −name '*.o' \) −atime +7 −exec rm {} \;
```

FILES

/etc/passwd, /etc/group

SEE ALSO

chmod(1), cpio(1), sh(1), test(1).
fs(4), stat(2) in the *3B2 Computer System Programmer Reference Manual.*

NAME

join − relational database operator

SYNOPSIS

join [options] file1 file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is −, the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line [see *sort*(1)].

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

The default input field separators are blank, tab, or new-line. In this case, multiple separators count as one field separator, and leading separators are ignored. The default output field separator is a blank.

Some of the below options use the argument *n*. This argument should be a **1** or a **2** referring to either *file1* or *file2*, respectively. The following options are recognized:

−a*n*	In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.
−e *s*	Replace empty output fields by string *s*.
−j*n m*	Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file. Fields are numbered starting with **1**.
−o *list*	Each output line comprises the fields specified in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number. The common field is not printed unless specifically requested.
−t*c*	Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant. The character *c* is used as the field separator for both input and output.

EXAMPLE

The following command line will join the password file and the group file, matching on the numeric group ID, and outputting the login name, the group name and the login directory. It is assumed that the files have been sorted in ASCII collating sequence on the group ID fields.

```
join −j1 4 −j2 3 −o 1.1 2.1 1.6 −t: /etc/passwd /etc/group
```

SEE ALSO

awk(1), comm(1), sort(1), uniq(1).

BUGS

With default field separation, the collating sequence is that of **sort −b**; with **−t**, the sequence is that of a plain sort.

The conventions of *join*, *sort*, *comm*, *uniq* and *awk*(1) are wildly incongruous.

Filenames that are numeric may cause conflict when the **-o** option is used right before listing filenames.

NAME

newform − change the format of a text file

SYNOPSIS

newform [**−s**] [**−i**tabspec] [**−o**tabspec] [**−b**n] [**−e**n] [**−p**n] [**−a**n] [**−f**] [**−c**char] [**−l**n] [files]

DESCRIPTION

Newform reads lines from the named *files*, or the standard input if no input file is named, and reproduces the lines on the standard output. Lines are reformatted in accordance with command line options in effect.

Except for **−s**, command line options may appear in any order, may be repeated, and may be intermingled with the optional *files*. Command line options are processed in the order specified. This means that option sequences like "**−e**15 **−l**60" will yield results different from "**−l**60 **−e**15". Options are applied to all *files* on the command line.

−s Shears off leading characters on each line up to the first tab and places up to 8 of the sheared characters at the end of the line. If more than 8 characters (not counting the first tab) are sheared, the eighth character is replaced by a * and any characters to the right of it are discarded. The first tab is always discarded.

An error message and program exit will occur if this option is used on a file without a tab on each line. The characters sheared off are saved internally until all other options specified are applied to that line. The characters are then added at the end of the processed line.

For example, to convert a file with leading digits, one or more tabs, and text on each line, to a file beginning with the text, all tabs after the first expanded to spaces, padded with spaces out to column 72 (or truncated to column 72), and the leading digits placed starting at column 73, the command would be:

```
newform −s −i −l −a −e file-name
```

−i*tabspec* Input tab specification: expands tabs to spaces, according to the tab specifications given. *Tabspec* recognizes all tab specification forms described in *tabs*(1). In addition, *tabspec* may be **−−**, in which *newform* assumes that the tab specification is to be found in the first line read from the standard input (see *fspec*(4)). If no *tabspec* is given, *tabspec* defaults to **−8**. A *tabspec* of **−0** expects no tabs; if any are found, they are treated as **−1**.

−o*tabspec* Output tab specification: replaces spaces by tabs, according to the tab specifications given. The tab specifications are the same as for **−i***tabspec*. If no *tabspec* is given, *tabspec* defaults to **−8**. A *tabspec* of **−0** means that no spaces will be converted to tabs on output.

−b*n* Truncate *n* characters from the beginning of the line when the line length is greater than the effective line length (see **−l***n*). Default is to truncate the number of characters necessary to obtain the effective line length.

The default value is used when **−b** with no *n* is used. This option can be used to delete the sequence numbers from a COBOL program as follows:
newform −l1 −b7 file-name

−e*n* Same as **−b***n* except that characters are truncated from the end of the line.

−p*n* Prefix *n* characters (see **−c***k*) to the beginning of a line when the line length is less than the effective line length. Default is to prefix the number of characters necessary to obtain the effective line length.

−a*n* Same as **−p***n* except characters are appended to the end of a line.

−f Write the tab specification format line on the standard output before any other lines are output. The tab specification format line which is printed will correspond to the format specified in the *last* **−o** option. If no **−o** option is specified, the line which is printed will contain the default specification of **−8**.

−c*k* Change the prefix/append character to *k*. Default character for *k* is a space.

−l*n* Set the effective line length to *n* characters. If *n* is not entered, **−l** defaults to 72. The default line length without the **−l** option is 80 characters. Note that tabs and backspaces are considered to be one character (use **−i** to expand tabs to spaces).

The **−l1** must be used to set the effective line length shorter than any existing line in the file so that the **−b** option is activated.

DIAGNOSTICS

All diagnostics are fatal.

usage: ...	*Newform* was called with a bad option.
not −s format	There was no tab on one line.
can't open file	Self-explanatory.
internal line too long	A line exceeds 512 characters after being expanded in the internal work buffer.
tabspec in error	A tab specification is incorrectly formatted, or specified tab stops are not ascending.
tabspec indirection illegal	A *tabspec* read from a file (or standard input) may not contain a *tabspec* referencing another file (or standard input).

0 — normal execution
1 — for any error

SEE ALSO

csplit(1), tabs(1).
fspec(4) in the *3B2 Computer System Programmer Reference Manual.*

BUGS

Newform normally only keeps track of physical characters; however, for the **−i** and **−o** options, *newform* will keep track of backspaces in order to line up tabs in the appropriate logical columns.

Newform will not prompt the user if a *tabspec* is to be read from the standard input (by use of **−i− −** or **−o− −**).

If the **−f** option is used, and the last **−o** option specified was **−o− −**, and was preceded by either a **−o− −** or a **−i− −**, the tab specification format line will be incorrect.

NAME

nl − line numbering filter

SYNOPSIS

nl [−**h**type] [−**b**type] [−**f**type] [−**v**start#] [−**i**incr] [−**p**] [−**l**num] [−**s**sep] [−**w**width] [−**n**format] [−**d**delim] file

DESCRIPTION

Nl reads lines from the named *file* or the standard input if no *file* is named and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

Nl views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (e.g., no numbering of header and footer lines while numbering blank lines only in the body).

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

Line contents	*Start of*
\:\:\:	header
\:\:	body
\:	footer

Unless optioned otherwise, *nl* assumes the text being read is in a single logical page body.

Command options may appear in any order and may be intermingled with an optional file name. Only one file may be named. The options are:

−**b***type* Specifies which logical page body lines are to be numbered. Recognized *types* and their meaning are:

−**h***type* Same as −**b***type* except for header. Default *type* for logical page header is **n** (no lines numbered).

a	number all lines
t	number lines with printable text only
n	no line numbering
p*string*	number only lines that contain the regular expression specified in *string*.

Default *type* for logical page body is **t** (text lines numbered).

−**f***type* Same as −**b***type* except for footer. Default for logical page footer is **n** (no lines numbered).

−**v***start#* *Start#* is the initial value used to number logical page lines. Default is **1**.

−**i***incr* *Incr* is the increment value used to number logical page lines. Default is **1**.

−**p** Do not restart numbering at logical page delimiters.

−**l***num* *Num* is the number of blank lines to be considered as one. For example, −**l2** results in only the second adjacent blank being numbered (if the appropriate −**ha**, −**ba**, and/or −**fa** option is set). Default is **1**.

−**s***sep* *Sep* is the character(s) used in separating the line number and the corresponding text line. Default *sep* is a tab.

−w*width* *Width* is the number of characters to be used for the line number. Default *width* is **6**.

−n*format* *Format* is the line numbering format. Recognized values are: **ln**, left justified, leading zeroes suppressed; **rn**, right justified, leading zeroes supressed; **rz**, right justified, leading zeroes kept. Default *format* is **rn** (right justified).

−d*xx* The delimiter characters specifying the start of a logical page section may be changed from the default characters (\:) to two user-specified characters. If only one character is entered, the second character remains the default character (:). No space should appear between the **−d** and the delimiter characters. To enter a backslash, use two backslashes.

EXAMPLE

The command:

```
nl −v10 −i10 −d!+ file1
```

will number file1 starting at line number 10 with an increment of ten. The logical page delimiters are !+.

SEE ALSO

pr(1).

NAME

od — octal dump

SYNOPSIS

od [**−bcdosx**] [file] [[**+**]offset[**.**][**b**]]

DESCRIPTION

Od dumps *file* in one or more formats as selected by the first argument. If the first argument is missing, **−o** is default. The meanings of the format options are:

−b	Interpret bytes in octal.
−c	Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null=**\0**, backspace=**\b**, form-feed=**\f**, new-line=**\n**, return=**\r**, tab=**\t**; others appear as 3-digit octal numbers.
−d	Interpret words in unsigned decimal.
−o	Interpret words in octal.
−s	Interpret 16-bit words in signed decimal.
−x	Interpret words in hex.

The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If **.** is appended, the offset is interpreted in decimal. If **b** is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded by **+**.

Dumping continues until end-of-file.

NAME

pack, pcat, unpack — compress and expand files

SYNOPSIS

pack [**–**] [**–f**] name ...

pcat name ...

unpack name ...

DESCRIPTION

Pack attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file *name* is replaced by a packed file *name*.**z** with the same access modes, access and modified dates, and owner as those of *name*. The **-f** option will force packing of *name*. This is useful for causing an entire directory to be packed even if some of the files will not benefit. If *pack* is successful, *name* will be removed. Packed files can be restored to their original form using *unpack* or *pcat*.

Pack uses Huffman (minimum redundancy) codes on a byte-by-byte basis. If the **–** argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of **–** in place of *name* will cause the internal flag to be set and reset.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each **.z** file, it is usually not worthwhile to pack files smaller than three blocks, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60-75% of their original size. Load modules, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90% of the original size.

Pack returns a value that is the number of files that it failed to compress.

No packing will occur if:

> the file appears to be already packed;
> the file name has more than 12 characters;
> the file has links;
> the file is a directory;
> the file cannot be opened;
> no disk storage blocks will be saved by packing;
> a file called *name*.**z** already exists;
> the **.z** file cannot be created;
> an I/O error occurred during processing.

The last segment of the file name must contain no more than 12 characters to allow space for the appended **.z** extension. Directories cannot be compressed.

Pcat does for packed files what *cat*(1) does for ordinary files, except that *pcat* cannot be used as a filter. The specified files are unpacked and written to the standard output. Thus to view a packed file named *name*.**z** use:

```
pcat name.z
```

or just:

```
pcat name
```

To make an unpacked copy, say *nnn*, of a packed file named *name*.**z** (without destroying *name*.**z**) use the command:

```
pcat name >nnn
```

Pcat returns the number of files it was unable to unpack. Failure may occur if:

the file name (exclusive of the **.z**) has more than 12 characters;
the file cannot be opened;
the file does not appear to be the output of *pack*.

Unpack expands files created by *pack*. For each file *name* specified in the command, a search is made for a file called *name*.**z** (or just *name*, if *name* ends in **.z**). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the **.z** suffix stripped from its name, and has the same access modes, access and modification dates, and owner as those of the packed file.

Unpack returns a value that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in *pcat*, as well as for the following:

a file with the "unpacked" name already exists;
if the unpacked file cannot be created.

SEE ALSO

cat(1).

NAME

paste − merge same lines of several files or subsequent lines of one file

SYNOPSIS

paste file1 file2 ...
paste **−d**list file1 file2 ...
paste −s [**−d**list] file1 file2 ...

DESCRIPTION

In the first two forms, *paste* concatenates corresponding lines of the given input files *file1*, *file2*, etc. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). If you will, it is the counterpart of *cat*(1) which concatenates vertically, i.e., one file after the other. In the last form above, *paste* replaces the function of an older command with the same name by combining subsequent lines of the input file (serial merging). In all cases, lines are glued together with the *tab* character, or with characters from an optionally specified *list*. Output is to the standard output, so it can be used as the start of a pipe, or as a filter, if − is used in place of a file name.

The meanings of the options are:

−d Without this option, the new-line characters of each but the last file (or last line in case of the **−s** option) are replaced by a *tab* character. This option allows replacing the *tab* character by one or more alternate characters (see below).

list One or more characters immediately following **−d** replace the default *tab* as the line concatenation character. The list is used circularly, i.e., when exhausted, it is reused. In parallel merging (i.e., no **−s** option), the lines from the last file are always terminated with a new-line character, not from the *list*. The list may contain the special escape sequences: **\n** (new-line), **\t** (tab), \ (backslash), and **\0** (empty string, not a null character). Quoting may be necessary, if characters have special meaning to the shell (e.g., to get one backslash, use *−d*"\\\\").

−s Merge subsequent lines rather than one from each input file. Use *tab* for concatenation, unless a *list* is specified with **−d** option. Regardless of the *list*, the very last character of the file is forced to be a new-line.

− May be used in place of any file name, to read a line from the standard input. (There is no prompting).

EXAMPLES

ls | paste −d" " −	list directory in one column
ls | paste − − − −	list directory in four columns
paste −s −d"\t\n" file	combine pairs of lines into lines

SEE ALSO

cut(1), grep(1), pr(1).

DIAGNOSTICS

line too long	Output lines are restricted to 511 characters.
too many files	Except for −s option, no more than 12 input files may be specified.

NAME

pg − file perusal filter for CRTs

SYNOPSIS

pg [−*number***] [−p** *string***] [−cefns] [+***linenumber***] [+/***pattern***/] [**files...**]**

DESCRIPTION

The *pg* command is a filter which allows the examination of *files* one screenful at a time on a CRT. (The file name − and/or NULL arguments indicate that *pg* should read from the standard input.) Each screenful is followed by a prompt. If the user types a carriage return, another page is displayed; other possibilities are enumerated below.

This command is different from previous paginators in that it allows you to back up and review something that has already passed. The method for doing this is explained below.

In order to determine terminal attributes, *pg* scans the *terminfo* (see the *3B2 Computer System Terminal Information Utilities Guide*) data base for the terminal type specified by the environment variable **TERM**. If **TERM** is not defined, the terminal type **dumb** is assumed.

The command line options are:

−*number*
An integer specifying the size (in lines) of the window that *pg* is to use instead of the default. (On a terminal containing 24 lines, the default window size is 23).

−p *string*
Causes *pg* to use *string* as the prompt. If the prompt string contains a "%d", the first occurrence of "%d" in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is ":".

−c Home the cursor and clear the screen before displaying each page. This option is ignored if **clear_screen** is not defined for this terminal type in the *terminfo* (see the *3B2 Computer System Terminal Information Utilities Guide*) data base.

−e Causes *pg* *not* to pause at the end of each file.

−f Normally, *pg* splits lines longer than the screen width, but some sequences of characters in the text being displayed (e.g., escape sequences for underlining) generate undesirable results. The −*f* option inhibits *pg* from splitting lines.

−n Normally, commands must be terminated by a <*newline*> character. This option causes an automatic end of command as soon as a command letter is entered.

−s Causes *pg* to print all messages and prompts in standout mode (usually inverse video).

+*linenumber*
Start up at *linenumber*.

+/*pattern*/
Start up at the first line containing the regular expression pattern.

The responses that may be typed when *pg* pauses can be divided into three categories: those causing further perusal, those that search, and those that modify the perusal environment.

Commands which cause further perusal normally take a preceding *address*, an optionally signed number indicating the point from which further text should be displayed. This *address* is interpreted in either pages or lines depending on the command. A signed *address* specifies a point relative to the current page or line, and an unsigned *address* specifies an address relative to the beginning of the file. Each command has a default address that is used if none is provided.

The perusal commands and their defaults are as follows:

(+1) <*newline*> or <*blank*>
: This causes one page to be displayed. The address is specified in pages.

(+1) **l** With a relative address this causes *pg* to simulate scrolling the screen, forward or backward, the number of lines specified. With an absolute address this command prints a screenful beginning at the specified line.

(+1) **d** or **^D**
: Simulates scrolling half a screen forward or backward.

The following perusal commands take no *address*.

. or **^L** Typing a single period causes the current page of text to be redisplayed.

$ Displays the last windowful in the file. Use with caution when the input is a pipe.

The following commands are available for searching for text patterns in the text. The regular expressions described in *ed*(1) are available. They must always be terminated by a <*newline*>, even if the −*n* option is specified.

i/*pattern*/
: Search forward for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately after the current page and continues to the end of the current file, without wrap-around.

i^*pattern*^
i?*pattern*?
: Search backwards for the *i*th (default *i*=1) occurrence of *pattern*. Searching begins immediately before the current page and continues to the beginning of the current file, without wrap-around. The ^ notation is useful for Adds 100 terminals which will not properly handle the ?.

After searching, *pg* will normally display the line found at the top of the screen. This can be modified by appending **m** or **b** to the search command to leave the line found in the middle or at the bottom of the window from now on. The suffix **t** can be used to restore the original situation.

The user of *pg* can modify the environment of perusal with the following commands:

*i***n** Begin perusing the *i*th next file in the command line. The *i* is an unsigned number, default value is 1.

*i***p** Begin perusing the *i*th previous file in the command line. *i* is an unsigned number, default is 1.

*i***w** Display another window of text. If *i* is present, set the window size to *i*.

s *filename*
: Save the input in the named file. Only the current file being perused is saved. The white space between the **s** and *filename* is optional. This command must always be terminated by a <*newline*>, even if the −*n* option is specified.

h Help by displaying an abbreviated summary of available commands.

q or Q Quit *pg*.

!*command*
: *Command* is passed to the shell, whose name is taken from the **SHELL** environment variable. If this is not available, the default shell is used. This command must always be terminated by a <*newline*>, even if the −*n* option is specified.

At any time when output is being sent to the terminal, the user can hit the quit key (normally control-\) or the interrupt (break) key. This causes *pg* to stop sending output, and display the prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

If the standard output is not a terminal, then *pg* acts just like *cat*(1), except that a header is printed before each file (if there is more than one).

EXAMPLE

A sample usage of *pg* in reading system news would be

```
news | pg -p "(Page %d):"
```

NOTES

While waiting for terminal input, *pg* responds to **BREAK**, **DEL**, and ^ by terminating execution. Between prompts, however, these signals interrupt *pg*'s current task and place the user in prompt mode. These should be used with caution when input is being read from a pipe, since an interrupt is likely to terminate the other commands in the pipeline.

Users of Berkeley's *more* will find that the z and f commands are available, and that the terminal /, ^, or ? may be omitted from the searching commands.

FILES

1027.sp40u

/usr/lib/terminfo/*
: Terminal information data base

/tmp/pg*
: Temporary file when input is from a pipe

SEE ALSO

ed(1), grep(1).
3B2 Computer System Terminal Information Utilities Guide.

BUGS

If terminal tabs are not set every eight positions, undesirable results may occur.

When using *pg* as a filter with another command that changes the terminal I/O options terminal settings may not be restored correctly.

NAME

sdiff − side-by-side difference program

SYNOPSIS

sdiff [options ...] file1 file2

DESCRIPTION

Sdiff uses the output of *diff*(1) to produce a side-by-side listing of two files indicating those lines that are different. Each line of the two files is printed with a blank gutter between them if the lines are identical, a < in the gutter if the line only exists in *file1*, a > in the gutter if the line only exists in *file2*, and a | for lines that are different.

For example:

```
x       |       y
a               a
b       <
c       <
d               d
        >       c
```

The following options exist:

−w *n*	Use the next argument, *n*, as the width of the output line. The default line length is 130 characters.
−l	Only print the left side of any lines that are identical.
−s	Do not print identical lines.
−o *output*	Use the next argument, *output*, as the name of a third file that is created as a user-controlled merging of *file1* and *file2*. Identical lines of *file1* and *file2* are copied to *output*. Sets of differences, as produced by *diff*(1), are printed; where a set of differences share a common gutter character. After printing each set of differences, *sdiff* prompts the user with a % and waits for one of the following user-typed commands:

l	append the left column to the output file
r	append the right column to the output file
s	turn on silent mode; do not print identical lines
v	turn off silent mode
e l	call the editor with the left column
e r	call the editor with the right column
e b	call the editor with the concatenation of left and right
e	call the editor with a zero length file
q	exit from the program

On exit from the editor, the resulting file is concatenated on the end of the *output* file.

SEE ALSO

diff(1), ed(1).

NAME

split — split a file into pieces

SYNOPSIS

split [−*n*] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000 lines) onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically, up to **zz** (a maximum of 676 files). *Name* cannot be longer than 12 characters. If no output name is given, **x** is default.

If no input file is given, or if − is given in its stead, then the standard input file is used.

SEE ALSO

bfs(1), csplit(1).

NAME

sum − print checksum and block count of a file

SYNOPSIS

sum [**−r**] file

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file. It is typically used to look for bad spots, or to validate a file communicated over some transmission line. The option **−r** causes an alternate algorithm to be used in computing the checksum.

SEE ALSO

wc(1).

DIAGNOSTICS

"Read error" is indistinguishable from end of file on most devices; check the block count.

NAME

tail − deliver the last part of a file

SYNOPSIS

tail [±[number][**lbc**[**f**]]] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +*number* from the beginning, or −*number* from the end of the input (if *number* is null, the value 10 is assumed). *Number* is counted in units of lines, blocks, or characters, according to the appended option **l**, **b**, or **c**. When no units are specified, counting is by lines.

With the **−f** ("follow") option, if the input file is not a pipe, the program will not terminate after the line of the input file has been copied, but will enter an endless loop, wherein it sleeps for a second and then attempts to read and copy further records from the input file. Thus it may be used to monitor the growth of a file that is being written by some other process. For example, the command:

```
tail −f fred
```

will print the last ten lines of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed. As another example, the command:

```
tail −15cf fred
```

will print the last 15 characters of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed.

SEE ALSO

dd(1M) in the *3B2 Computer System Administration Utilities Guide.*

BUGS

Tails relative to the end of the file are stored in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tr − translate characters

SYNOPSIS

tr [**−cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. Any combination of the options **−cds** may be used:

−c Complements the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 001 through 377 octal.

−d Deletes all input characters in *string1*.

−s Squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[a−z] Stands for the string of characters whose ASCII codes run from character **a** to character **z**, inclusive.

[a∗*n*] Stands for *n* repetitions of **a**. If the first digit of *n* is **0**, *n* is considered octal; otherwise, *n* is taken to be decimal. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2*.

The escape character \ may be used as in the shell to remove special meaning from any character in a string. In addition, \ followed by 1, 2, or 3 octal digits stands for the character whose ASCII code is given by those digits.

EXAMPLE

The following example creates a list of all the words in *file1* one per line in *file2*, where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the shell; 012 is the ASCII code for newline.

```
tr −cs "[A−Z][a−z]" "[\012∗]" <file1 >file2
```

SEE ALSO

ed(1), sh(1).
ascii(5) in the *3B2 Computer System Programmer Reference Manual.*

BUGS

Will not handle ASCII NUL in *string1* or *string2*; always deletes NUL from input.

NAME

uniq − report repeated lines in a file

SYNOPSIS

uniq [**−udc** [**+**n] [**−**n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. *Input* and *output* should always be different. Note that repeated lines must be adjacent in order to be found; see *sort*(1). If the **−u** flag is used, just the lines that are not repeated in the original file are output. The **−d** option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the **−u** and **−d** mode outputs.

The **−c** option supersedes **−u** and **−d** and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The *n* arguments specify skipping an initial portion of each line in the comparison:

−*n*	The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.
+*n*	The first *n* characters are ignored. Fields are skipped before characters.

SEE ALSO

comm(1), sort(1).

Index